Forces of Nature

ANGLE: An Experimental Test-bed for Graph Embedders and the Big Bang: a New Force Directed Embedder

Alan Creek

BSc Honours Research Report
Department of Computer Science
University of Canterbury

November 6, 2001

Supervisor: Dr. Neville Churcher

Abstract

Many systems in the modern world are readily represented as graphs; with nodes representing objects within the system, and edges the relationships between them. The effectiveness of visualisations of complex systems is often dependent on the layout of their graphs; a good layout can provide ready understanding of complexity, while a poor layout does nothing to aid comprehension and can even obscure information. We examine the use of force directed placement (FDP) as an algorithm for automatically laying out connected graphs and introduce ANGLE, a software tool for experimentation with graph layout. FDP assigns properties to nodes and edges analogous with real world objects; principally treating edges as springs and nodes as bodies that are mutually repellent. Iterating over the graph and positioning the nodes according to the notional forces upon them provides a means to arrive at a layout. Previous work has focused primarily on two dimensional layouts with various force models. We tackle the extension into three dimensions and propose a new force model that produces a more reliable quality of layout in a shorter time than earlier models. In addition we look at the problem of terminating layout algorithms at an appropriate time, and examine several means of terminating algorithms when a suitable end result has been achieved.

Contents

List of Figures

1 Introduction

Understanding complexity is an increasingly recurrent theme in modern science. This is particularly true in the field of software development where the source code can be both huge and distributed across many files. Program code, and many other documents, often naturally lend themselves to representation as graphs; where nodes represent 'objects' within the document and edges represent the relationships between them. Examples from software engineering include UML diagrams, parse trees and call graphs; from other disciplines graphs represent transportation and computer networks, organisational charts, PERT charts and so on. Understanding of such documents can be greatly aided by some form of visualisation [24], but is contingent on the graph being laid out in a readily understandable way. Our focus is on computer software, but the issue is a general one.

Tools have been developed over the years to aid in the process of describing and understanding complex documents. These range from diagramming techniques such as Unified Modelling Language (UML), through to various tools that provide graphical representations of system structure. Automatic methods for generating diagrammatic representations are myriad, but effective tools for laying out diagrams, even in two dimensions, are less common. In any case the level of complexity encountered in modern software is such that any two-dimensional rendering must, of necessity, fail to convey a sufficient level of detail if it is to be comprehensible at all. Three dimensional (3D) renderings of graphs should be of considerable benefit [39; 27; 18]. The third dimension provides a greatly increased information space and the 3D view is a natural one for human interpretation. The development of the Virtual Reality Modelling Language (VRML) and concomitant web browser plug-ins has provided us with an effective, readily available, means of viewing and manipulating 3D structures (Figure 1.1). Suitably arranging arbitrary graph structures in 3D space is less well supported.

Many algorithms exist for the automatic layout of graph drawings, for the most part in two dimensions (2D). Several of these algorithms use some variant of 'force directed placement' (FDP) where the nodes of the graph are imagined to repel each other while, at the same time, being drawn together by spring-like edges [14]. FDP promises to provide a means of solving otherwise intractable layout problems. It is essentially a means of applying numerical optimisation to heuristically solve the famed n-body problem of Newtonian physics. Unlike similar methodologies used by physical scientists however, we seek aesthetically pleasing solutions rather than faithful representations of real structures. This gives us the freedom to manipulate the model and the algorithm we use, in order to improve efficiency of computation and quality of layout.

A software tool for determining desirable layouts of drawings is usually referred to as an 'embedder'. An embedder typically consists of an algorithm, a model, and some controlling parameters. In our work with FDP we have aimed to construct a general embedder to efficiently produce aesthetically pleasing, readily understandable layouts of graphs. A good embedder should be independent of the structure of the graph being

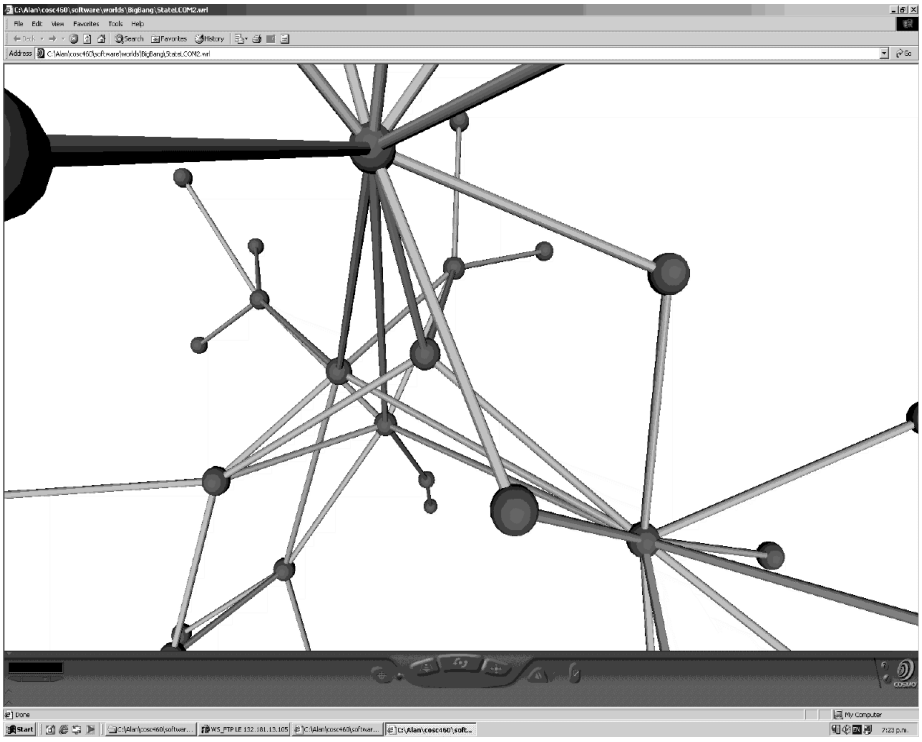


Figure 1.1: Viewing a 3D graph with a web browser VRML plug-in

laid out, and should produce consistent results on a given graph regardless of initial configuration. Since our ultimate target is to create 3D VRML worlds we have made no attempt to constrain layouts within any spatial bounds or to produce 3D drawings that project nicely into 2D. Our purpose is to fully utilise the extra information space of 3D to provide effective visualisations of complex data sets.

To facilitate this research we have implemented an experimental toolkit providing both graphical and command line applications for analysing algorithm and model behaviour, and capturing statistical data on the layout process. The software works equally well with 2D or 3D layouts and is capable of writing VRML world files for display and browsing via VRML browsers. We have implemented 3D versions of some well known embedders and propose a new method, the Big Bang, using a two phase model that addresses most of the current issues with existing FDP systems.

In this report we review some of the previous developments in graph layout methods in Section 2 and describe the development and implementation of our experimental software in Section 3. In Section 4 we give an overview of some existing layout algorithms, discuss force models for FDP and examine computational complexity. In the latter part of Section 4 we introduce our force model and compare it with some others, then discuss the extension of FDP to 3D. This is followed, in Section 5, by a discussion of methods of terminating FDP algorithms and the associated issues.

Section 6 describes and discusses the Big Bang model and reports our results from its application to a variety of graph types. We then offer some conclusions in Section 7 and finally close with a section on future work.

2 Related work

2.1 Force directed placement

Previous work on FDP dates back at least as far as 1963 when Tutte published his "How to Draw a Graph", that presented some basic theorems for both 2D and 3D drawing and introduced the 'barycentre' concept for creating convex layouts [38]. For our purposes, however, research into graph drawing began with Eades [9], after the advent of the desktop computer when the desire for rapid 'interactive' layout of graphs evolved. Work since is largely based on the heuristics for quality proposed by Eades; namely minimising edge crossings in the layout and preserving symmetry where it occurs. Some doubt on the value of the edge crossing measure has been expressed by Kamada and Kawai [25] and more recent work has tended to aim for evenness of edge length. Whether edge length is an appropriate measure of layout quality is arguable, but we need some criterion, and large variance in edge length in a drawing intuitively produces poor layouts.

Previous FDP methods implement minor variations on Eades' original algorithm and vary the force models used somewhat. We expand on this and review some force models in Section 4.3. Algorithm development has focused largely on adding enhancements to combat undesirable behaviour under certain conditions of Eades original. Some have concentrated on the problem of unfavourable initial arrangements that produce poor layouts. This has been addressed either by adding nodes one at a time to a layout [36; 13; 15], or by encouraging nodes stuck in unfavourable locations to 'leap past' the nodes that trap them [14; 8]. Others have tackled the problem of when to terminate an algorithm. FDP is an iterative process and requires some means of determining when a layout is satisfactory and the algorithm should terminate. This generally involves using some sort of global measure of layout quality such as a minimal energy configuration [25; 8] or a cost function that terminates when a pre-determined value is attained [3; 14]. A cost function is designed to indicate the effort required to improve a given layout, if the 'cost' is too high the layout is considered complete. Since FDP models tend towards an equilibrium state most algorithms can simply be run until equilibrium is reached [9; 11]. This approach is often wasteful, however, as a satisfactory layout may be achieved well before equilibrium occurs.

In Section 4 we give an overview of the evolution of FDP since 1983. For an extensive overview of layout algorithms see [2] and more recent updates in [10]. The first part of Section 5 reviews some of the termination techniques that have been used.

2.2 3D graph layout

Almost all previous work relates to 2D graph layout. Fruchterman and Reingold [14] discuss the use of their embedder to lay out 3D shapes, but their aim is to produce

suitable 2D projections of inherently 3D graphs. Cohen et al. [7] discuss methods for generating 3D layouts, concentrating on orthogonal drawings of trees and minimisation of the space consumed. They examine the general 3D layout problem and propose theoretical performance limits for the process.

In other work the third dimension is applied to an essentially 2D layout as a means of encoding some attribute of the graph nodes. Reiss [32] proposes a method for presenting program call graphs in a 3D space. This is, however, a strongly application specific approach that, like many others, uses the third dimension to render non-spatial data rather than to provide true 3D layouts. In Gil and Kent [18] inter-related 2D diagrams are drawn on the $x-z$ plane and stacked vertically with relations drawn as edges through 3D space; again, this is an application specific technique. Another approach is that of Gogolla et al. [19] where UML diagrams are laid out in 3D space and the perspective produced is used to provide a focus+context view of the data.

In the specific area of laying out graphs in 3D, Kumar and Fowler [28] detail a 3D version of Kamada and Kawai's algorithm [25]. The original algorithm involves solving a system of linear equations, with partial derivatives of the x and y ordinates as coefficients, for each node placement. The extension to 3D produces computations that are truly prodigious.

Gajer et al. [15] discusses a multi-dimensional embedder combining parts of several other techniques to produce rapid layout of large graphs. Their algorithm is complex and based on energy minimisation with 'intelligent' initial placement using a "vertex filtration" function to apply a divide and conquer approach to layout. By their own admission their method falls down somewhat with highly connected graphs, such as might be expected in the domain within which we are working.

2.3 Experimental software

Reiss [32] used an object oriented (OO) framework to produce a 3D layout tool for tree structures based on OO software metrics (call graphs, class diagrams, etc.). The focus was on software visualisation rather than graph layout, but the framework approach provides flexibility for implementing different layout methodologies, and the graphical interface provides animated views of layouts as they are created.

GraphEd [21] is an experimental tool that provides graph editing and visualisation in 2D and allows users to code their own graph manipulation functions using its 'Sgraph' data structures. The primary focus is on the presentation of laid out graphs, rather than experimentation with algorithms and models. The application is written in C, but uses the concept of modular design with appropriate separation of concerns between inputting graphs, graph layout, presentation and output. The interface effectively places the various models within a single, user friendly, interface. A successor to GraphEd, GraphLet is now a part of an ongoing research project at the University of Passau. GraphLet, however, has become a complex mix of Tcl/Tk and C++ with a steep learning curve to follow for effective use of the tool.

In 1999 'LayoutShow', a signed Java applet, appeared [4]. The tool features animations of layout operation, standardised format of input and output files, and runtime configuration of algorithm parameters. It also has the ability to generate randomised initial layouts and has an easy to use and effective user interface. For our purposes LayoutShow falls short in that its design does not allow for easy extension (addition of new algorithms or models) and provides insufficient separation of concerns between layout algorithms and methods of termination. It is also limited to 2D.

Patrignani and Vargiu [30] developed '3DCube', a tool designed for very similar purposes to ours. However, the software is not readily available and is designed primarily for orthogonal or grid based layouts.

Many other implementations of experimental tools exist; [33] provides an overview of some of the better known ones. Generally, however, existing layout software has been produced by researchers to serve their specific purposes, usually as a byproduct of their own work, rather than as generic experimental test-beds. For our work we required something of a 'Swiss army knife', that could be used to implement any algorithm or force model we wished to work with. We needed an application that could evolve its capabilities to keep pace with the changing demands of our research without constant re-engineering of the software.

3 ANGLE

'Alan and Neville's Graph Layout Experimenter' (ANGLE) is an application we have developed for the purpose of implementing and experimenting with graph layout methodologies. We begin this section by stating our requirements and follow with an overview the design and implementation process of ANGLE. We then discuss ANGLE's user interfaces and some of its features.

3.1 Requirements

The general process of laying out a graph is depicted in Figure 3.1. In creating ANGLE we needed to implement this process and provide means of evaluating performance. Our principle requirements were:

- A graphical display of layouts produced,
- animations of the progress of embedders during the layout process,
- the ability to readily implement different algorithms, models and layout terminators,
- the ability to configure embedder parameters at runtime and to load and run any combination of algorithm, model and terminator,
- the ability to add additional input and output formats for graphs and output of experimental data in any desired format with minimal effort,
- production of multiple runs from randomised input so as to gather statistical data on embedder performance and behaviour,
- understandability of the software and its operation, allowing others to continue the work with minimal learning overhead.

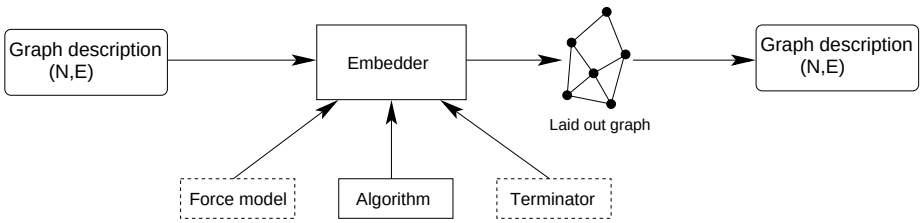


Figure 3.1: The graph layout process

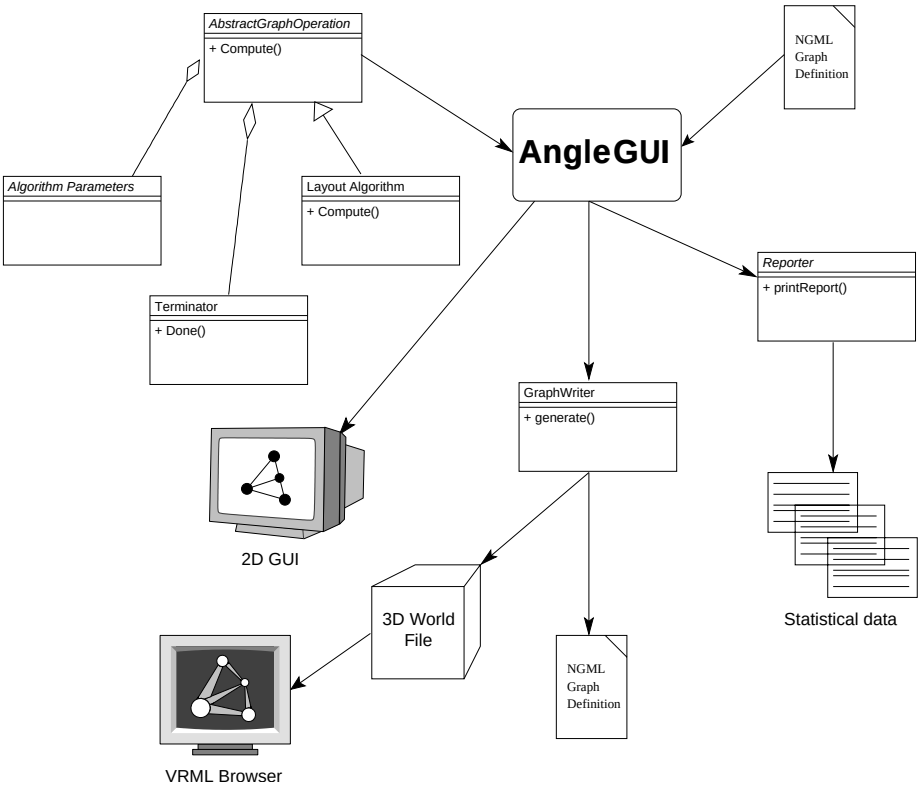


Figure 3.2: ANGLE architecture

While speed of operation was not considered critical, layout of graphs with hundreds of nodes in a reasonable time was required. Our intention for performance evaluation was to concentrate on minimising the number of iterations of an algorithm rather than to maximise the execution speed of a single iteration.

3.2 Design and implementation

The general architecture of ANGLE is depicted in Figure 3.2.

In order to provide maximum flexibility ANGLE has been designed as a collection of cohesive units with minimal coupling between them, using well understood object oriented (OO) techniques [34]. This supported our desire for code reuse, extensibility and maintainability. We chose Java as our implementation language; its 'write once run anywhere' claim may be arguable but it does produce highly portable software, and it supports the OO paradigm well. The ability to implement new classes without the need to re-compile the entire application, or even have access to the source code, was also seen as an advantage [29]. Additionally, the rich application program interface (API) and class library of the Java software development kit (SDK) meant that much of the non-application specific functionality could be handled by using existing classes. This greatly eased development, allowing us to concentrate on the job in hand and not get 'bogged down' with 'housekeeping' matters. Java's Javadoc facility was also

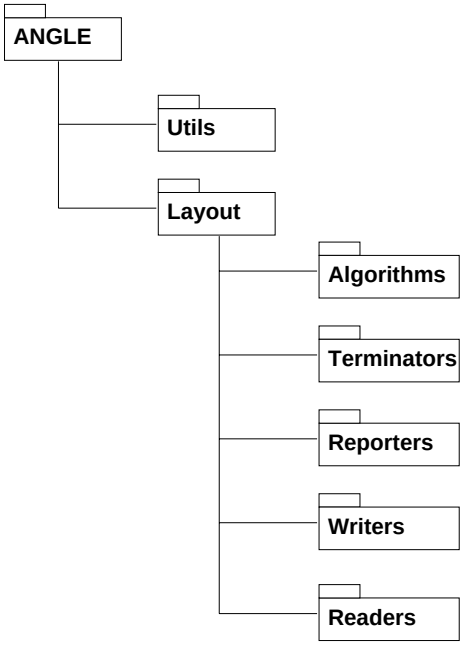


Figure 3.3: ANGLE package structure

a factor in the selection of the language. Javadoc provides a useful tool for creating maintenance and re-use documentation and ANGLE has been extensively 'Javadoced'. The final product consists of 70 Java classes, some 8,500 lines of code, and 2,800 lines of Javadoc. This amounts to remarkably little coding given the functionality and features of the application.

Broadly the architecture may be divided into graph input, algorithms and models, terminators, graph output, and data output. Each of these five units is reliant upon ANGLE's internal representation of graphs, but is otherwise capable of independent operation. The Java package structure of the implementation is shown in Figure 3.3. The utils package contains useful, but not graph oriented, classes, while the layout package contains all of the generalised data handling and framework classes. The five packages below layout, on the right of the diagram, contain functional classes, each of which implements an appropriate interface, or abstract class, from the layout package.

Like GraphEd (and GraphLet) we wanted an easily extensible system so that algorithms and models could be rapidly implemented or changed at will. Using OO encapsulation via the strategy design pattern [16] we have removed the need for the user to know any details of the data structure used to represent graphs and have provided interface classes for users to implement without the need for understanding of the underlying mechanics. The interface of our graph class provides views of a graph as either a collection of nodes or a collection of edges. This allows implementors to construct algorithms based on either, or both, views with equal facility.

In addition to the interfaces for the five component types we have provided abstract classes that implement all of the required 'housekeeping' functionality. By extending these classes, a user wishing to implement, say, a new algorithm has only one method to write. Runtime control of algorithm or terminator parameters is achieved by having the

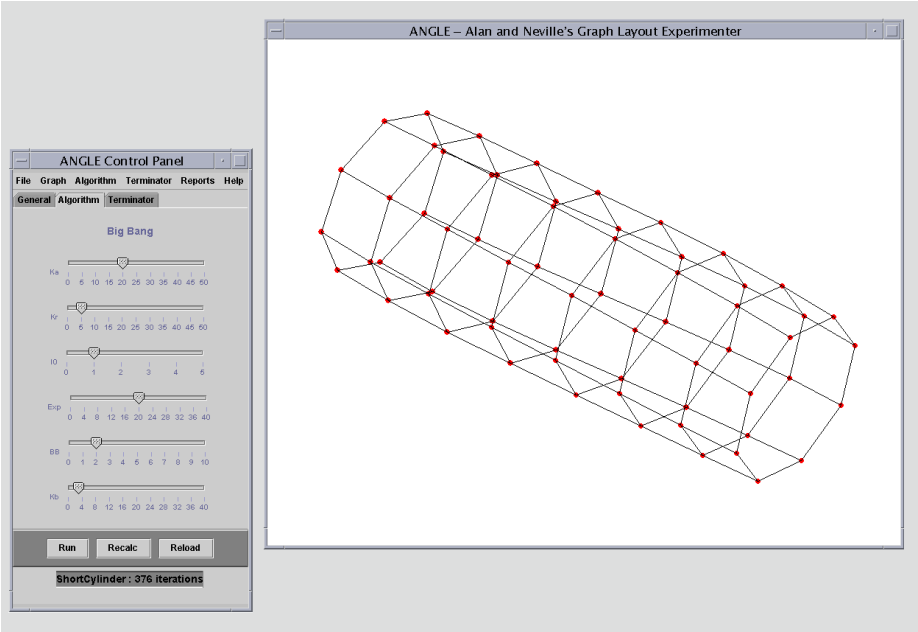


Figure 3.4: ANGLE graphical interface in action

algorithm or terminator class implement a collection of control objects. This is handled by the abstract classes so an implementing user need only specify the controls required and write the code to act on the values received at runtime. In all cases the interfaces we have designed have been kept as general as possible; it should be possible to take any concrete class from ANGLE and use it without alteration in another application. ANGLE can therefore serve as a development and test bed for layout algorithms to be used in other software.

3.3 Graphical interface

ANGLEGUI, the graphical user interface (GUI), ties all five components together in a user friendly application. The main menu for the GUI is created at startup based on the concrete implementations of the main units found in pre-defined Java packages. This application of an abstract factory-like design pattern [16] allows for the creation of new classes in any of the main units without the need to alter any of the existing code. As an additional aid to extensibility the Java class that contains the GUI control panel implements the observable design pattern [16] by way of allowing the addition of Java ActionListeners that are triggered by any user interaction with the GUI.

It was decided to implement only a simple (2D) graphical display and provide output in VRML format for full 3D viewing. VRML is well supported by several web-browser plug-ins providing full functionality for 'flying' or 'walking' through laid out graphs [5]. The 2D interface gives a fast, interactive, display suitable for quick evaluation of a layout that can then be written to a VRML file if more detailed examination is desired.

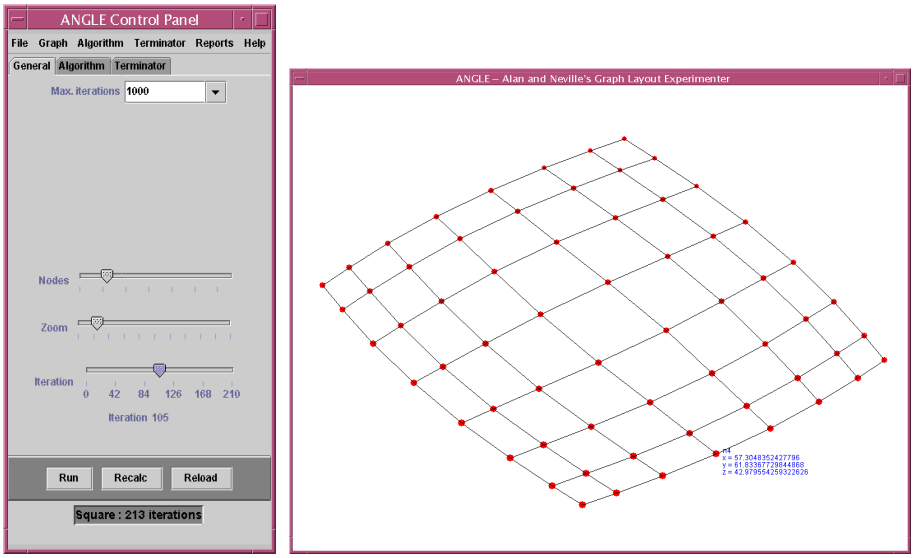
A screen shot of ANGLE's GUI in action is shown in Figure 3.4. Here the user

has used the Big Bang algorithm to lay out a cylindrical graph; the control panel is displaying the tabbed pane for algorithm configuration–the Big Bang class provides six configurable parameters. This pane is customised interactively from the algorithm object when the user makes a selection from the algorithm menu.

Figure 3.5(a) shows the 'General' pane of the control panel. The 'Max. iterations' control sets a number of iterations after which the current algorithm will be forcefully terminated (irrespective of any selected terminator). The 'Node' slider control sets the size that nodes are drawn at in the display, and the 'Zoom' slider set the total drawing size. These may both be set interactively. The 'Iteration' slider selects the iteration of the algorithm that is displayed. In the figure the user has selected iteration number 105.

Independent of the tabbed pane, and therefore visible at all times, are the three buttons and the label beneath them. The 'Run' button causes the iteration slider to be set to zero, and then steps it through a tick at a time to the current maximum value. As the slider steps, each iteration is drawn in in the display window, thus giving an automatic animation of the layout process. The 'Recalc' button recalculates the layout using current settings; this is used to update a layout after changing any algorithm or terminator parameter settings. The 'Reload' button reloads the graph in a new random arrangement and calculates a layout for it. Finally the label at the bottom gives the title of the current graph (from the NGML description) and the number of iterations used in the last layout calculation.

The display window uses a naive approach to present some visual cues concerning the layout. Nodes in 3D layouts are sized in proportion to their z-axis ordinate, with large z values producing smaller nodes. Individual nodes may be identified by pointing and clicking with the mouse, whereupon ANGLEGUI displays the node's name and its coordinates in 3D space. These two features are shown in Figure 3.5(b).



(a) 'General' pane (b) Square grid layout

Figure 3.5: ANGLEGUI having laid out a 64 node square grid

Finally, ANGLEGUI interprets any value assigned to nodes in the NGML input file as RGB colour specifications, rendering nodes accordingly. This is a useful feature when implementing algorithms that make use of nodes' 'value' fields for layout purposes; nodes of like colour should behave similarly.

For the purpose of studying algorithm behaviour we decided to compute an entire graph layout and history and retain this in memory. This approach provides the ability to interactively scroll back and forth through the iterations of an algorithm and view, or review, the progress of the layout. It also allows the implementation of retrospective data gathering. For example we have implemented a 'distance to run' reporter that produces a trace of the mean node distance from final position.

3.4 Command line interface

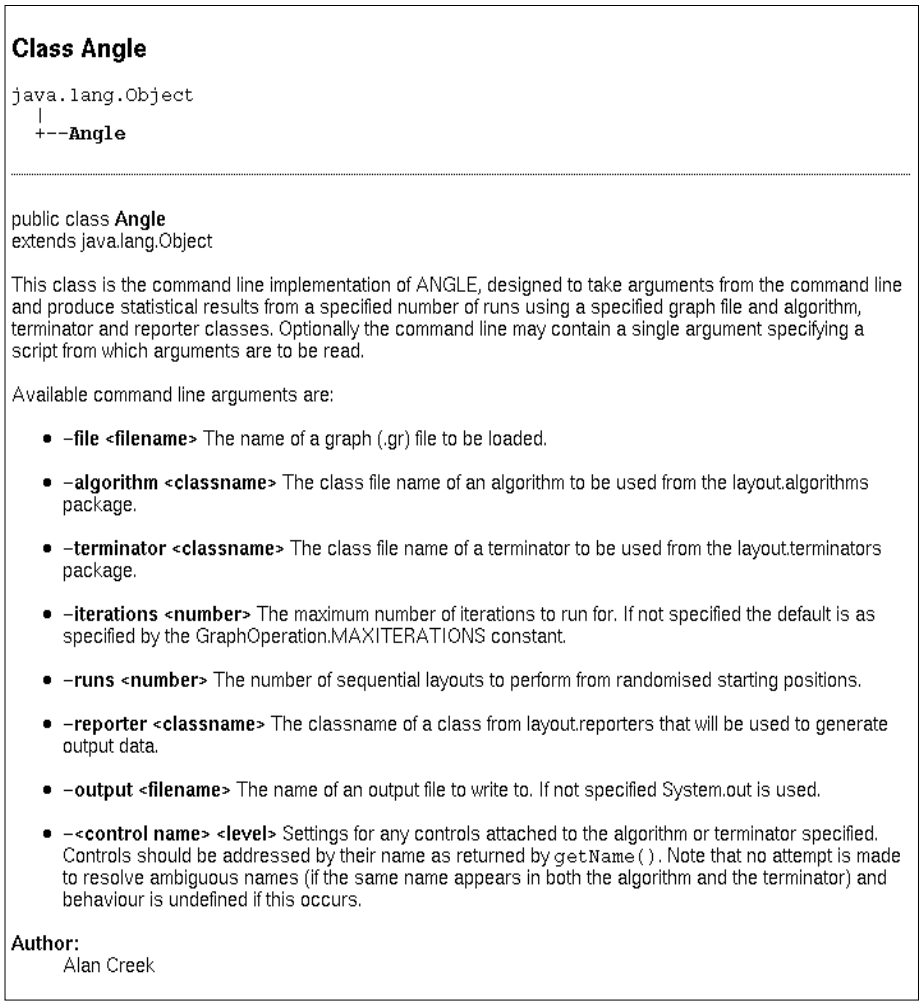
In addition to the GUI application we have also implemented a text based, command line driven application that can be used to run many embeddings on randomised copies of a given graph, and output statistical data on an embedder's behaviour. The command line version is fully configurable, either by command line arguments or by script file. A sample script, used to gather some of the data for this report, is shown in Figure 3.6. Figure 3.7 shows a part of the Javadoc documentation for using the class, explaining the command line arguments available.

3.5 Execution speed

Speed of operation is not considered critical, but execution should not proceed too slowly. Java is essentially an interpreted language and we found its performance a hindrance at times, particularly with larger graphs or more complex embedders. Some tests were conducted to determine where bottlenecks may occur, the results of which are summarised in Table 3.1. The results shown are mean times in milliseconds for one hundred runs of one million operations, although the iterations are the mean time to iterate over a collection or array of one million objects. Of probable significance

```
# Edge length trace script
#
file gr/gridcylindertall.gr
algorithm BigBang
# set BB iterations to 2N
# accept de-
faults for other parameters
BB 2
terminator NoMovement
# termination limit
limit 5
reporter EdgeTrace
runs 50
output data/BB_tallcyl_edgetrace.dat
summary
iterations 5000
```

Figure 3.6: ANGLE command line version configuration script



Class Angle

```
java.lang.Object
   |
   +--Angle
```

public class **Angle**
extends java.lang.Object

This class is the command line implementation of ANGLE, designed to take arguments from the command line and produce statistical results from a specified number of runs using a specified graph file and algorithm, terminator and reporter classes. Optionally the command line may contain a single argument specifying a script from which arguments are to be read.

Available command line arguments are:

- **–file <filename>** The name of a graph (.gr) file to be loaded.
- **–algorithm <classname>** The class file name of an algorithm to be used from the layout.algorithms package.
- **–terminator <classname>** The class file name of a terminator to be used from the layout.terminators package.
- **–iterations <number>** The maximum number of iterations to run for. If not specified the default is as specified by the GraphOperation.MAXITERATIONS constant.
- **–runs <number>** The number of sequential layouts to perform from randomised starting positions.
- **–reporter <classname>** The classname of a class from layout.reporters that will be used to generate output data.
- **–output <filename>** The name of an output file to write to. If not specified System.out is used.
- **–<control name> <level>** Settings for any controls attached to the algorithm or terminator specified. Controls should be addressed by their name as returned by `getName()`. Note that no attempt is made to resolve ambiguous names (if the same name appears in both the algorithm and the terminator) and behaviour is undefined if this occurs.

Author:
Alan Creek

Figure 3.7: ANGLE command line application, Javadoc of options

to algorithm implementors is the overhead incurred in making method calls. We noticed this particularly in our original algorithm implementation where we had several method calls for purposes of calculation. An execution speed increase of some 300% was achieved by including the mathematics in the layout method itself. Note also that floating point division is much faster than integer division and that floating point operations are at least as fast as integer operations in other cases. ANGLE uses the Java `double` type for all numeric values.

Table 3.1 also indicates that iteration over arrays is much faster than iteration through collections. This is probably at least partially a consequence of the two method calls required for each collection access when iterating through a collection. Despite this we have used Java collections in our implementation, although this decision may need to be reviewed at a later date.

Operation	time (ms)
Create objects	90
Iterate collection	245
Iterate array	42
Method call	35
FP multiply	9
FP divide	22
Int multiply	9
Int divide	63

Table 3.1: Java times for various operations

3.6 NGML

'Neville's Graph Markup Language' (NGML) is a system for encoding graphs in simple text files, based on the Extensible Markup Language (XML). The code produced is easily parsed by machine while retaining a reasonable level of human readability [6]. The language provides markup tags for describing multiple graphs in a single file, with an optional title for each graph. Nodes have a name, and may have location and floating point attributes in addition. Edges are described simply by defining their source and destination nodes. The NGML code of a graph with the connectivity of a tetrahedron is shown in Figure 3.8. We have used NGML as the input mechanism for ANGLE, and as one of the available output formats. We have found it satisfactory for this purpose, and particularly useful when the need to add an enhancement to the language has arisen. Several further language additions have already been proposed, some of these are discussed in Section 8.

Our current parser for NGML was written using Java CUP and JFlex [22; 26]. While this approach makes writing parsers quite easy and fast, it does mean that users must have the Java CUP class library installed.

Herman and Marshall [20] describe a more sophisticated approach using a true XML language format designed for graph expression. It may prove realistic to abandon NGML and implement a parser for ANGLE for this format. Sun Microsystems now provide a Java class library and specification for processing XML files. Use of this API could obviate the Java CUP/JFlex parser, making the software more portable.

3.7 Summary

ANGLE provides both a toolkit for working with graph layout and, through the GUI and command line executables, a versatile and readily extendible test-bed for algorithm, force model, and terminator experiments. Its ability to both read and write NGML files allows the user to save specific graph configurations. This provides repeatability of layout runs and the ability to 'freeze' specific initial arrangements of interest for later work. To achieve this the output NGML files always contain `<coord>` tags specifying each node's location in the layout.

Finally, ANGLE, and the algorithms we have implemented thus far, all behave exactly as for a 2D implementation if the z-axis coordinates of all nodes in a graph are set to zero. This has proven a convenient way of comparing our layout results with previous work that has only been carried out in 2D.

```
#
# Generated by ANGLE v1.0
#
# Fri Oct 19 12:11:25 GMT+13:00 2001
#
<graphset>
<graph><title>Tetrahedron</title>
<nodes>
#Node nC
<node><value>0xff0000</value><name>nC</name>
<coord>7.3525E0 72.807E0 48.487E0</coord></node>

#Node nA
<node><value>0xff</value><name>nA</name>
<coord>22.294E0 62.152E0 72.245E0</coord></node>

#Node nB
<node><value>0xff00</value><name>nB</name>
<coord>76.636E0 75.132E0 72.63E0</coord></node>

#Node nD
<node><value>0xffff00</value><name>nD</name>
<coord>45.107E0 23.189E0 47.696E0</coord></node>

</nodes>

<edges>
<edge><from>nC</from><to>nD</to></edge>
<edge><from>nA</from><to>nD</to></edge>
<edge><from>nA</from><to>nC</to></edge>
<edge><from>nB</from><to>nD</to></edge>
<edge><from>nB</from><to>nC</to></edge>
<edge><from>nA</from><to>nB</to></edge>
</edges>
</graph>
</graphset>
```

Figure 3.8: NGML code of a tetrahedral graph

4 Layout algorithms

A graph G consists of a set of nodes, or vertices N, and a set of edges E, where each edge is an unordered pair of nodes (u, v), defining connections between nodes. An edge may have $u = v$, which is a self loop, or may appear more than once in a graph (a multiple edge) [10]. Following convention in graph layout work, we constrain ourselves to 'simple graphs', that is graphs without self loops or multiple edges. In addition we consider only connected graphs, those where all nodes have at least one edge. For the purposes of layout a non-connected graph can be considered as multiple connected graphs so the extension to include non-connected graphs should not prove difficult.

Eades [9], following from previous work on printed circuit board layout [31], first proposed a 'spring-based' graph layout methodology in 1983. The principle behind this class of layout algorithms is to consider the nodes of a graph as rings or balls that are connected by edges modelled as steel springs, as shown in Figure 4.1. The idea being that if the nodes are assumed to be mutually repulsive there should be a 'natural' equilibrium state between this repulsion and the attraction of the springs. The arrangement at equilibrium is expected to produce an aesthetically pleasing and efficient layout.

In general, spring algorithms fall broadly into two categories: force directed and energy minimisation. Some mixing of the two does occur where a force directed approach uses some form of total energy function to measure 'completeness' of a layout in order to terminate an algorithm. Examples of this can be found in Behzadi [3]; Kamada and Kawai [25] and Kumar and Fowler [28].

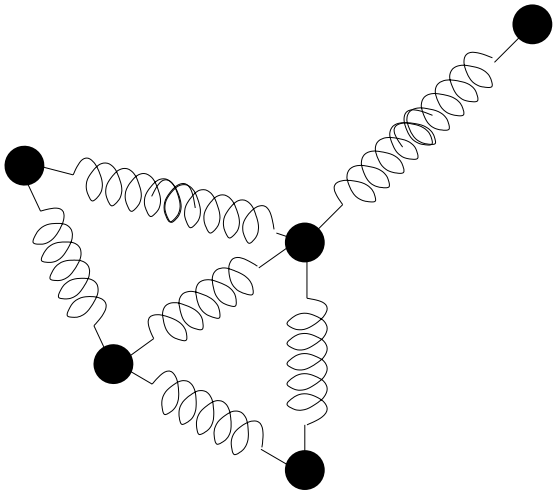


Figure 4.1: A ball and spring model of a graph

```
CONSTANT K_a            // Constant of attraction
CONSTANT K_r            // Constant of repulsion

layout(Graph G)
    place nodes of G at random locations
    repeat until done
        for each node n in G
            set repulsive force vector (F_r) on n to zero
            for each other node m ≠ n in G
                F_r = F_r+ repulsive force vector of m on n
            set attractive force vector (F_a) on n to zero
            for each neighbour m of n
                F_a = F_a+ attractive force vector of m on n
            set net force vector (F_n) on n to K_a × F_a − K_r × F_r
        move each node by F_n
        evaluate termination condition
```

Figure 4.2: The basic force directed algorithm

Energy minimisation algorithms operate by considering the arrangement of nodes and edges in a graph to represent some form of potential energy, typically the energy of springs under tension, and lay out a graph by moving nodes at each step to reduce the energy level until some desired minimum is reached. These algorithms require energy level measurement over some set of nodes for the calculation of each node's movement at each iteration. In some cases the entire graph is taken into consideration, but more typically a 'local minimum' is sought for each node and the total graph energy is considered only once all localities have been minimised. This approach means that the algorithms are inclined to be computationally expensive; Kamada and Kawai, for example, solve a set of partial differential equations for each displacement made. An alternate strategy is to 'push' a node in a random, or semi-random direction, and only actually displace it if energy reduction is achieved; Davidson and Harel [8] use this approach in their simulated annealing model. With these algorithms a node movement may be calculated and then not be executed if the result would not decrease the total energy. This is disadvantageous in that significant computation may occur that produces no improvement in layout.

Force directed placement (FDP), on the other hand, utilises the strength and direction of the net force on a node to determine its direction and amount of movement at each step. Figure 4.2 lists the pseudo-code for a simple force directed algorithm.

Unlike energy minimisation algorithms, there is no implicit termination scheme for FDP. Algorithms may be terminated by total energy measurement, or by some other technique. Termination techniques for FDP algorithms are discussed in Section 5.

4.1 Review of FDP

Eades [9] used a logarithmic function for his spring force and only implemented repulsion between pairs of non-neighbours. Others have commented that this system moved each node as its displacement was calculated, rather than moving all nodes 'at once' after completing all calculations. We find little evidence in Eades' original paper to support this claim. In any case, experimentation has shown that the effect on the

final result is minimal. An updated version used in Eades et al. [11] used linear spring forces and repulsion between *all* nodes.

Fruchterman and Reingold [14] followed essentially the algorithm of Figure 4.2 in their 'FR' embedder. They used a single 'constant' (representing the desired edge length) that reduced over time. This ensured termination of the algorithm but required a pre-determined 'cooling rate'. Their force formulae also differed from that of Eades, chiefly in the removal of the logarithmic function to improve computation. In the 'grid square variant' of their algorithm they also proposed a system of approximating repulsive forces by only considering nodes in close proximity.

Frick et al. [13] in their GEM algorithm, followed Eades' system of moving each node as its displacement was calculated, but introduced the concept of having an individual 'temperature' and cooling schedule for each node. This algorithm uses probabilistic methods and knowledge of each node's history in order to determine future movement. It also utilises attraction to a centre of gravity or 'barycentre' in addition to the spring and repulsive forces. The authors claim that GEM is faster than either [14] or [25], but still acknowledge a time complexity of $\mathcal{O}(|N|^3)$.

Kamada and Kawai [25] presented an embedder that use only spring forces and calculated ideal distances between nodes based on the graph theoretic distances between them. Their algorithm involved computing an all pairs shortest path and then solving linear equations for node positioning. It is not strictly an FDP approach and involves a higher complexity of mathematics than most others.

Davidson and Harel [8] introduced an algorithm using 'simulated annealing' (DH). At each iteration a new position for each node is proposed at a random location at a set distance r from the node's current position. The node is then moved only if a cost function indicates that an improvement would result. The value of r is steadily reduced as the algorithm progresses.

Tunkelang [36] proposed two algorithms, the first of which orders the nodes by determining a minimum height spanning tree for the graph. The nodes are then positioned in the layout, in order, with local optimisation based on previously placed nodes. This local positioning is then refined once all nodes have been placed and then a final fine tuning is done on the graph as a whole. Positioning is determined using Davidson and Harel's cost function.

In his second algorithm Tunkelang uses a modified version of the grid variant of Fruchterman and Reingold's algorithm. In this version the repulsive force approximation uses the method proposed by Barnes and Hut [1] and self-terminates when the mean square of node displacements in an iteration falls below a specified value.

Gansner and North [17] aimed to reduce clutter in graph layouts. They use Kamada and Kawai's algorithm to produce an initial layout and follow this with an expansive phase to obtain 'sufficient' distance between nodes. They then draw the edges as curves to avoid edge/node collisions and minimise crossings.

Behzadi [3] used a modified version of Fruchterman and Reingold's algorithm with an enhanced cost function and two phase layout. The first phase allows large movements of nodes to produce an approximate solution, while the latter phase is a fine tuning of the layout.

Gajer et al. [15] produced a 3D version of Kamada and Kawai [25]'s algorithm. This version also uses the technique of adding nodes one at a time to the layout.

Overall FDP algorithms are easy to implement and provide a simple metaphor that lends itself to easy understanding of the effect and performance of the algorithm. Tunkelang [36] noted, however, that "they are too slow" and "do not produce consistently good drawings". Our experimentation has borne this out to a large extent and we note several specific issues with FDP algorithms in general:

- They do not guarantee improvement at every iteration. Iterations during which the layout is not improved are wastage as far as the goal is concerned.
- All require appropriate settings of some initial values. In some cases this means selection of force constants; others require pre-setting of control parameters such as cooling rates, initial temperatures or other constraints.
- The results obtained are invariably dependent upon the initial positioning of the nodes. The level of dependency varies somewhat, but none of the algorithms listed above will reliably produce the same layout for a given graph regardless of starting position. Tunkelang [36] comments "in practice, the output quality of existing algorithms is inconsistent for general graphs".
- The amount of computation required increases in rough proportion to the reliability and quality of the resulting layout. This is largely a result of the fact that many algorithms have added complexity to attempt solutions to specific problems. For example GEM combats oscillation and rotation by comparing a node's movement with that of the previous iteration.

4.2 Algorithm complexity

To lay out a graph using an FDP algorithm we need to perform multiple iterations (I) of the algorithm of Figure 4.2, or something like it. Calculation of attractive forces involves examining every edge in the graph and can be performed in $\mathcal{O}(E_c)$ time in each iteration, where E_c is the number of edges in the graph. The repulsive force on a node, however, is determined as a function of every other node in the graph. It therefore requires $\mathcal{O}(N_c(N_c-1))$ time to calculate (where N_c is the number of nodes in the graph) in each iteration. The computational complexity of a standard FDP algorithm is thus $\mathcal{O}(IN_c(N_c-1))$.

Experience shows that I for any layout is invariably much greater than N_c and is linear in N, with the result that the complexity of FDP algorithms reduces to $\mathcal{O}(N_c^3)$. Other approaches to graph layout fare no better. Kamada and Kawai [25] do not iterate over the graph in the same manner but their nested looping still results in $\mathcal{O}(N_c^3)$. Davidson and Harel [8] in their simulated annealing variant claim $\mathcal{O}(N_c^2 E_c)$, but this is based on fixing the number of positioning trials, usually considered as linear in N_c, as a constant–a somewhat questionable notion.

At least two approaches to reducing complexity have been tried. Both attempt to limit the number of nodes considered in repulsive force calculations by ignoring pairs that are distant from each other. This makes sense when using an inverse square rule and should have no significant effect on results. Fruchterman and Reingold [14] propose a method to achieve this that assigns nodes to squares in a grid, and considers only nodes within the same grid square. This should prove adequate for fairly sparse

graphs, but may be less desirable for dense graphs where the (cumulative) repulsion of distant nodes is significant.

An alternative approach is to 'cluster' distant nodes and calculate repulsion from the barycentre of the cluster. Barnes and Hut [1] propose a force model of this type that gives $\mathcal{O}(N_c log N_c)$ for repulsive force calculations. This has the advantage of including the repulsive effect of *all* nodes, without having to explicitly calculate it.

While limiting the number of nodes considered in force calculations undoubtedly holds the promise of reducing computational complexity, the likely benefits are limited to larger graphs. In graphs with N_c less than a few hundred the extra cost of assigning nodes to clusters or grid squares is, in practice, likely to outweigh the gain from reduced complexity. In a practical implementation of graph layout we are concerned with run time rather than complexity, and can make significant gains by using computationally efficient calculations within each iteration, by minimising the number of iterations required, or by some combination of these two. For graphs of the size we consider in this report, this latter approach is more realistic.

4.3 Force models

All FDP algorithms require a force model. The model is used by the algorithm to calculate the net force on each node at each iteration. In actuality we model a unit of time with a single iteration of an algorithm and use the force model to calculate the *displacement* of each node, rather than the actual net force. The scalar displacement of a node in a single iteration is thus computed as the sum of at least two force functions (Equation 4.1).

$$D_n = \sum_{m \in N, m \neq n} F_r(n,m) + \sum_{(n,m) \in E} F_a(n,m) \tag{4.1}$$

A good force model should consist of computationally efficient formulae, follow a simple metaphor to aid human understanding, and follow 'sensible' force-distance relationships so that aesthetically pleasing layouts can be produced in a reasonable time.

In most cases FDP algorithms have used Hooke's law to provide a linear force-distance relationship for attraction and an inverse square relationship, after the electromagnetic interaction, for repulsion. Variants of both of these have been used; Eades used a logarithmic spring for example, and Fruchterman and Reingold used force formulae specifically designed to speed up computation. Other, more complex, models have been tried such as that of Kamada and Kawai [25] which solves a system of linear equations for each node displacement.

Another crucial factor in the force model is the notion of a preferred length or 'rest length' for the springs. This represents the desired length of edges in the final graph layout and models the length of a spring when no force is extant. The use of a rest length aids the final result aesthetically by ensuring a reasonable separation between nodes, and computationally by preventing collapse to a point. Some models explicitly incorporate rest length in their formulae [14; 9], while others rely on the balance between forces to ensure an appropriate edge length [36]. Table 4.1 shows the attractive and repulsive force calculations used by some of the better known algorithms.

Many force models utilise some notion of 'temperature' and 'cooling' designed to produce relatively large node movements early in the process; reducing to ever smaller movements as the layout nears completion, [14; 3; 15; 8; 13] for example. The chief

Algorithm	attraction	repulsion
Eades (1984)	$k_{a1} \times log_2(\frac{\delta}{k_{a2}})$	$\frac{k_r}{\delta^2}$
Eades (1997)	$k_a \times \frac{\delta - l_0}{\delta}$	$\frac{k_r}{\delta^2}$
Fruchterman & Reingold	$\frac{\delta^2}{l_0}$	$\frac{l_0^2}{\delta}$
Frick et al.	$\frac{\delta^2}{l_0^2(1+deg(v)/2)}$	$\frac{l_0^2}{\delta}$
Tunkelang (1994)	$k_a \times \delta^2$	$\frac{k_r}{\delta^2}$

δ = distance between nodes, l_0 = rest length of spring, k_n = force constant

Table 4.1: Force formulae for some algorithms

disadvantage of such models is that to calibrate the cooling effect it is necessary to have a reasonable notion as to the number of iterations required *before* producing the layout. We contend that a suitable force model should converge to a 'nice' layout as a natural consequence of the forces' interactions. Such a force model would also naturally terminate once a desirable layout has been produced, and thus operate robustly and reliably without prior knowledge of the layout requirements.

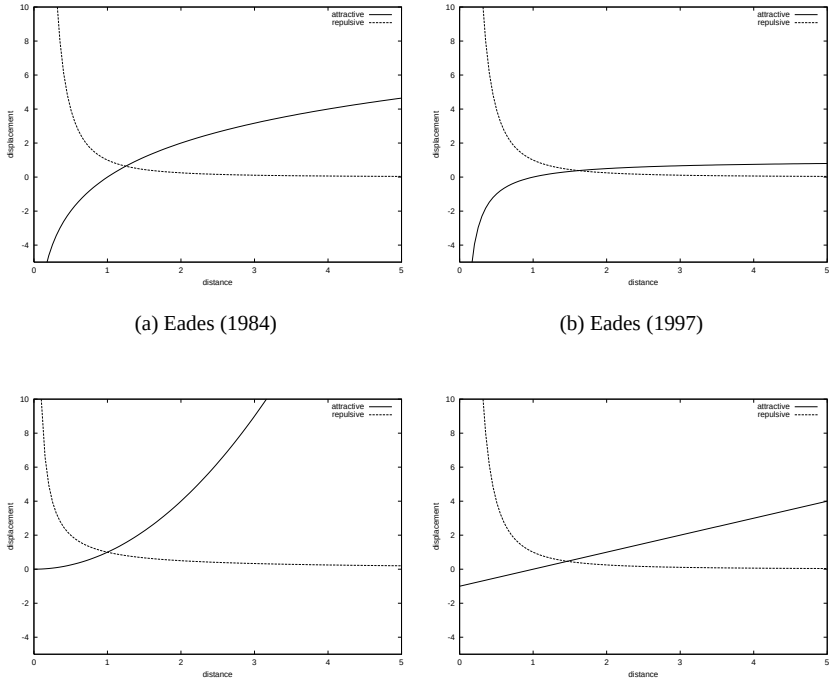
4.4 Our model (CC)

Aiming for simplicity, we chose to implement the algorithm of Figure 4.2 in our 'CC' (after Creek and Churcher) model. This is a minimal modification of Eades' original, that updates node positions all at once when displacements have been calculated. This method of updating node positions is the most commonly used, and more closely models the real world, aiding understanding of algorithm behaviour. For the force model we used two simple formulae, also slight modifications of the original. The attractive force is a standard Hooke's law spring force implemented so that no force is exerted when the spring is at its rest length. Repulsion is by way of a standard inverse square relationship. The formulae are shown in Equations 4.2 and 4.3.

$$F_a = k_a \times (\delta - l_0) \tag{4.2}$$

$$F_r = k_r \times \frac{1}{\delta^2} \tag{4.3}$$

The functional forms of CC are compared with others in Figure 4.3. Note that with l_0 (rest length of springs) set to 1 Tunkelang's forces are identical to those of Fruchterman and Reingold. Frick et al.'s model is also similar to Fruchterman and Reingold's, but with an additional term in the denominator of the attractive force function. This has the effect of flattening the attractive force curve in proportion to the degree of a node. All models have their equilibrium point ($F_a = F_r$) at approximately the rest length of a spring. In any graph with more than two nodes the ideal rest length will



(a) Eades (1984)

(b) Eades (1997)

(c) Fruchterman & Reingold

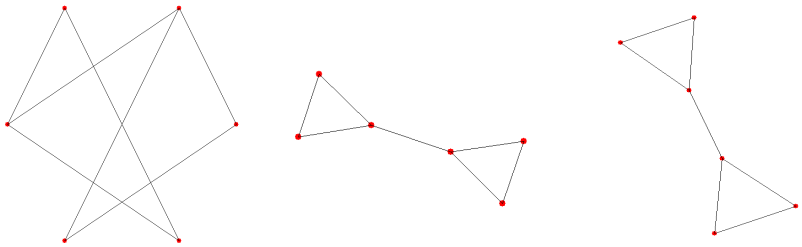
(d) Creek & Churcher

Figure 4.3: Functional forms of force models

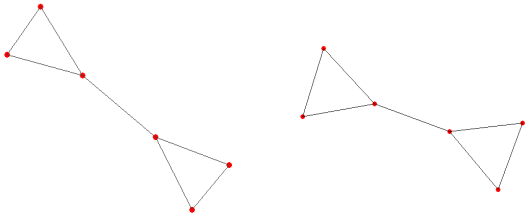
not be achieved so a reasonable approximation serves. CC is most like that of Eades [9], although Eades et al. [11] used a relatively large attractive force constant (3.0) that produces very similar curves. The others all use a d^2 term in their attractive force function, giving exponentially increasing attraction with distance. It may be this, possibly excessive, long range attraction that produces the sometimes wild oscillations we have observed with these models.

With our implementation of CC we obtained results comparable with most others for 2D layouts. Specific examples are presented in Figures 4.4, 4.5 and 4.6, that compare our algorithm and model with those of [9; 14; 25]. Eades did not specify the base for his logarithmic calculations. We have used log_2 in our implementation which produces results consistent with those in his paper. Figure 4.4 shows a graph from Figure 2 of Kamada and Kawai [25]. The layouts were generated from the starting position shown in Figure 4.4(a) as used by Kamada and Kawai. Figure 4.5 compares the same algorithms applied to a more complex graph. This graph was used by Kamada and Kawai [25] as an example where minimising edge crossings was unlikely to produce a readily comprehensible layout. The graph consists of five tetrahedral shapes where the outer four connect to the vertices of the inner one. In 2D this would ideally be laid out as five squares with diagonals across them. All four layouts are good, but the CC model produces more regular shaping of the outer shapes.

In Figure 4.6 we show layouts of the tree structure used by Eades [9]. There is little to choose between these results in terms of layout quality. The CC model is



(a) Initial layout (b) Eades' 1984 (c) Kamada & Kawai



(d) Fructerman & Reingold (e) Creek & Churcher

Figure 4.4: Layouts of the graph of Figure 2 from Kamada and Kawai [25]

significantly faster, however.

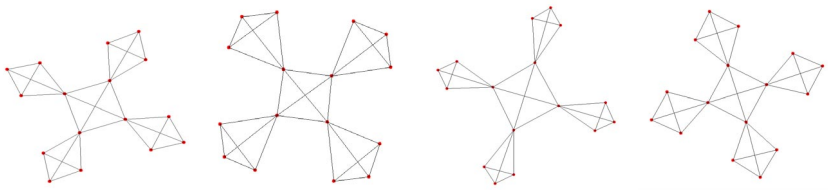
We used randomised initial positions in generating the layouts of Figures 4.4, 4.5, and 4.6, and found that all algorithms' results were somewhat dependent on the starting point. The diagrams show the 'best' layouts selected from ten random starting points.

In terms of the time taken to produce the sample layouts, Table 4.2 gives the number of iterations of each algorithm required to give the layouts of Figures 4.4, 4.5 and 4.6.

Previous papers have noted that FDP generally gives poor results on highly connected graphs or graphs with more than around 50 nodes. Behzadi [3] produced good results for regular shaped graphs up to around 250 nodes, but at the cost of a rather complex algorithm; others have done rather less well. Figure 4.7 shows typical layouts from Eades', Fruchterman and Reingold's and our CC system for a 100 node regular

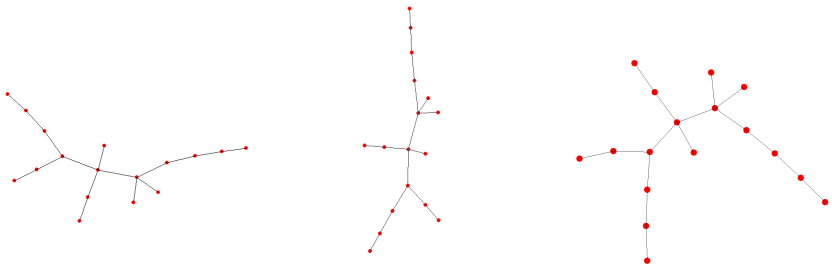
Layout	Algorithm		
	Eades	FR	CC
Fig. 4.4	83	92	33
Fig. 4.5	86	70	70
Fig. 4.6	237	100	120

Table 4.2: Iteration count comparisons



(a) Eades' 1984 (b) Kamada & Kawai (c) Fructerman and Reingold (d) Creek & Churcher

Figure 4.5: Layouts of the graph of Figure 1 from Kamada and Kawai [25]

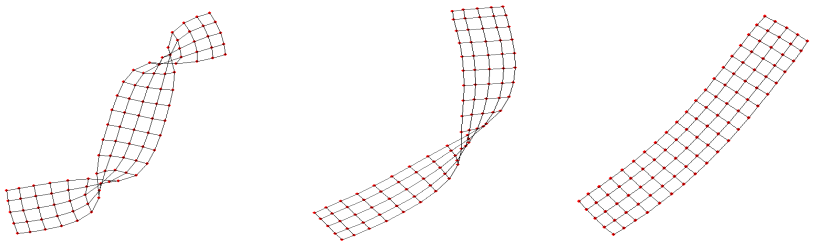


(a) Eades' 1984 (b) Fructerman & Reingold (c) Creek & Churcher

Figure 4.6: Layouts of the graph from Figure 4(b) of Eades [9]

grid. These three layouts consumed 705 (Eades), 400 (FR) and 798 (CC) iterations from the same starting configuration. FR produces good layouts (similar to that shown for CC) approximately half the time from a randomised start, requiring 800–1000 iterations to do so. CC produces about the same proportion of successes, though usually in 600–700 iterations. Behzadi, in her paper, consumed 500 iterations and produced a somewhat skewed grid with diamond shaped faces.

Overall, our very simple force model and algorithm combination gives layouts that are at least as good as those produced by others, and with less computational effort. Additionally the behaviour of the algorithm during layout production is considerably smoother than that for Eades or FR. In the iteration by iteration animations produced by ANGLE, our system smoothly evolves a layout, showing clear improvement at every iteration. As a consequence, if execution time is critical, our system can be terminated much sooner than others and still give a reasonable result. Consequently, all results from CC and the Big Bang in the remainder of this report were compiled using our simple 'NoMovement' terminator with the limit parameter set to 0.005. We discuss termination in greater detail in Section 5.



(a) Eades' 1984

(b) Fructerman & Reingold

(c) Creek & Churcher

Figure 4.7: Layouts of a 100 node (5x20) grid

4.5 Extension to 3D

Extending existing algorithms and models to the 3D case has not proven difficult. For the most part minor adjustments to the force models to accommodate the third ordinate of node positions is sufficient; producing layouts of comparable quality to the 2D versions.

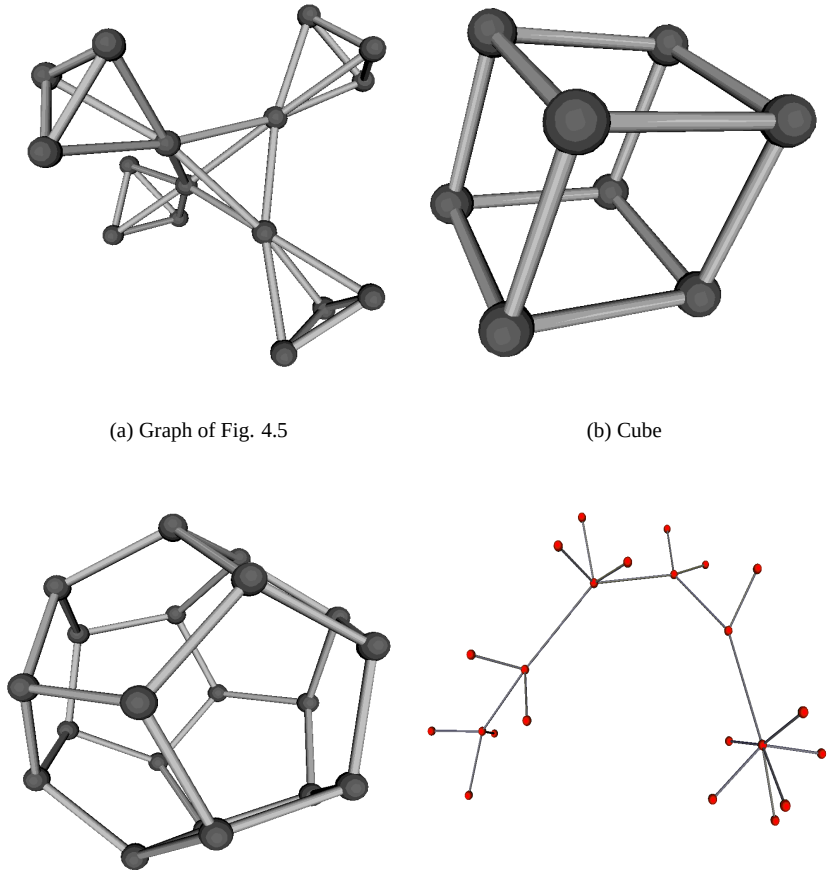
There are, however, some differences in the requirements when we are laying out graphs with the benefit of a third dimension. Many previous algorithms have sought to minimise edge crossings as an aesthetic criterion. This is of considerably less concern in the 3D case since the probability of an edge crossing occurring is greatly reduced. Kamada and Kawai [25] provide some evidence for edge crossing being an inappropriate, or at least unmanageable, criterion for even the 2D case.

Additionally, previous researchers have commented on problems that may occur when a node is 'trapped' on the wrong side of another, [14; 8] and others. This makes a 'nice' layout difficult or impossible to obtain. The extension to 3D serves to virtually eliminate this problem also, since these traps generally fail to occur.

A 3D implementation of a 2D algorithm can be expected to be about 50% more expensive in terms of computation of node positions. This effect should, to a large extent, be offset by being able to forgo added complexity used in avoiding edge crossings and node traps. Applying the CC system to graphs with known ideal shapes we have achieved excellent results in approximately the same time (number of iterations) as the 2D case. Some examples are presented in Figure 4.8.

4.6 Force constants

All FDP models use force constants as part of the formulae for determining displacement of nodes in each iteration. The force formulae of Table 4.1 show the constants used in some common models. Eades [9] uses a total of four constants, two for attraction, one for repulsion and an overall constant for setting the final displacement. At the other extreme FR uses the same single constant for both attraction and repulsion; achieving cooling by reducing the 'constant' value. Fruchterman and Reingold [14] consider their constant to be representative of spring rest length. Our experience shows that changing the constant has little effect on layout results other than scaling;



(a) Graph of Fig. 4.5

(b) Cube

(c) Dodecahedron

(d) Tree

Figure 4.8: 3D layouts using the CC embedder

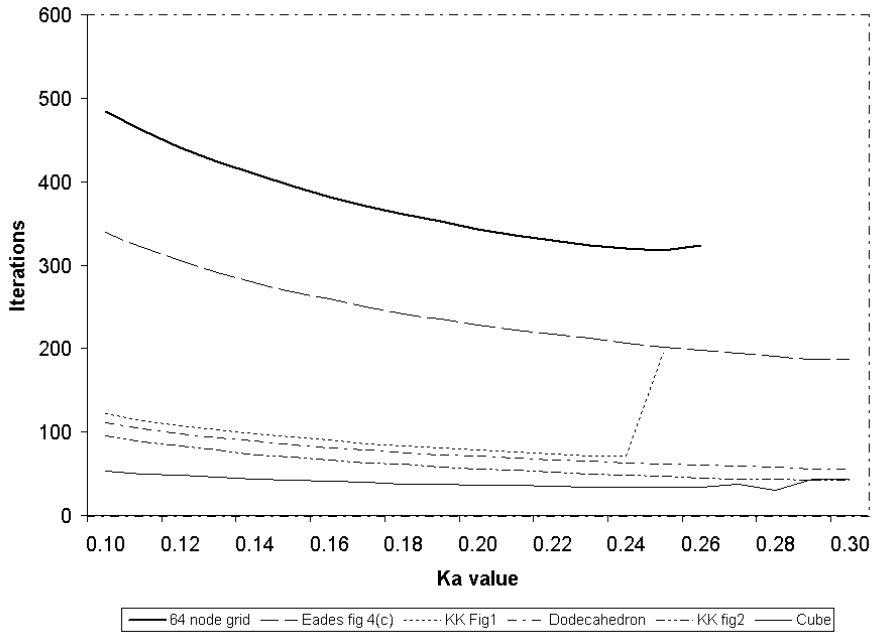


Figure 4.9: Effect of attraction constant value on iterations required

confirming the author's claim. Eades [9] used values of 2.0, 1.0, 1.0 and 0.1 for his four constants (effectively nullifying the second and third constants), and these values appear to give reasonable results for most graphs. The performance of Eades' system is rather more dependent on the choice of constants than is FR however. Eades notes that for graphs with near co-linear nodes he needed to adjust the constants to achieve a reasonable layout.

Experimenting with CC we found that best results were obtained with the attractive constant large relative to the repulsive one. Also, the number of iterations required depended most strongly on the attractive constant, and the quality of the final layout was found to be roughly in inverse proportion to the number of iterations. This is convenient in that optimising the number of iterations tends to produce the best layouts.

From the graph of Figure 4.9 it is evident that a K_a value of 0.24 will, in general, provide good results. With values greater than this amount the behaviour becomes unpredictable for many graphs; no graph that we have tried shows significant improvement for higher values of K_a. The K_r constant has much less effect on overall results. Its value as a proportion of the K_a constant appears more important and we observed best results for values between 0.04 and 0.06; roughly one fifth of the attractive constant). These results are valid for both the 2D and 3D cases.

5 Terminating layouts

Terminating the iteration of a graph layout algorithm at an appropriate time is a theme that runs through most previous research. Finding a suitable solution for a given algorithm requires assessing aesthetic criteria for the layout as it progresses and halting the process once the layout is 'good enough'. This presents the dual problems of finding appropriate measures of completeness and then assessing their value in a computationally efficient way.

Several termination techniques have been applied and broadly fall into three categories:

1. Use a cost function or other global measure of graph layout and terminate when a pre-determined minimum (or maximum) is reached, examples appear in [37; 3; 8; 25; 13], or
2. run the algorithm for a pre-determined number of iterations, [9; 14], or
3. stop the algorithm once all activity has ceased (i.e. when the nodes do not move for two or more successive iterations).

The first category of termination is a logical choice for energy minimisation algorithms. These generally use an energy measurement function in each iteration in any case, so terminating once a minimum is reached involves little added computation. With FDP algorithms, however, evaluating a cost function at each iteration can be expensive. Behzadi [3] uses a cost function as part of the determination of node movement, but requires significant additional calculation for evaluating termination conditions. Our experiments, using ANGLE, with cost function based termination, have indicated that the 'correct' termination time depends on both the characteristics of the graph and the initial arrangement of the nodes. Using Behzadi's 'CostSpring' function we found it necessary to re-calibrate the terminator for each type of graph. Even then the results were highly variable for any given graph over several randomised starting positions.

The second category, pre-determining the number of iterations, has obvious limitations. For most graphs, and certainly for an arbitrary graph, the number of iterations required cannot be adequately determined in advance. Fruchterman and Reingold [14] do not actually pre-determine the number of iterations, but rather the cooling rate. This has the same effect in their implementation, but does provide some flexibility. For example the cooling rate could be determined by the maximum movement in an iteration.

The third category of termination method is a special case of the first. The algorithm terminates once all node movement ceases; this is simply another global measure. This method assumes that the algorithm will, in fact, reach a point where no further updating of node positions is occurring. Furthermore, it assumes that the layout is suitable at this time and not significantly earlier. As an example, this method does not work well with FR; FR tends to arrive at a reasonable layout fairly quickly, and then oscillate with

decreasing amplitude as the system cools. It is the eventual cessation of movement through cooling that stops the algorithm, usually somewhat later than optimal. Given a well behaved algorithm and force model (one that converges smoothly to a final state), termination by lack of activity should work well.

Applying termination techniques to CC we have found that the simple 'No Movement' version works as well as any, and better than most. In actual operating layout programs this termination scheme can be implemented at negligible cost; the maximum movement in an iteration can be recorded while iterating over the nodes during force calculation. In practice 'no movement' would be defined as 'maximum movement less than a certain level' and layout time can be decreased (at some expense in quality) by simply increasing the 'certain level' used.

Applying various termination methods to CC we obtained typical results as depicted in Figure 5.1. The chart shows a trace of mean node movement in each iteration for the layout of a 64 node grid square and the stopping points for four terminators:

(a) CostSpring : Behzadi's 'CostSpring' function used to measure layout quality and terminate once a set value is reached. Our calibration set the limit as 0.0008 and used alpha = 0.5 [3].

(b) SpringTension : our version of a global measure of energy. This measures residual tension in the springs and takes an average. Termination occurs when the mean spring tension changes by less than the limit between two consecutive iterations. The calibration process set the limit to 0.01.

(c) NoMovement : measures the movement of each node in each iteration and terminates when the average falls below a specified value, in this case the limit set was 0.006.

(d) EdgeLength : measures the length of the edges and terminates when the change in average value over the last iteration drops below a specified value. The limit value set was 0.001 in this experiment.

In gathering the data we first used a randomised arrangement to calibrate the parameters for the four terminators tested so that all four stopped at approximately the same time (i.e. gave similar results). We then generated a second randomised starting position and applied each of the four calibrated terminators–these are the results shown in Figure 5.1 and produced the layouts of Figure 5.2. Plainly, for the CC embedder, the No Movement termination scheme gives best results. Both Behzadi's CostSpring and our SpringTension method terminate too soon, while the EdgeLength method terminates too late. These results are typical, although the EdgeLength terminator generally terminates when layout quality is reasonable it frequently uses more iterations than are required.

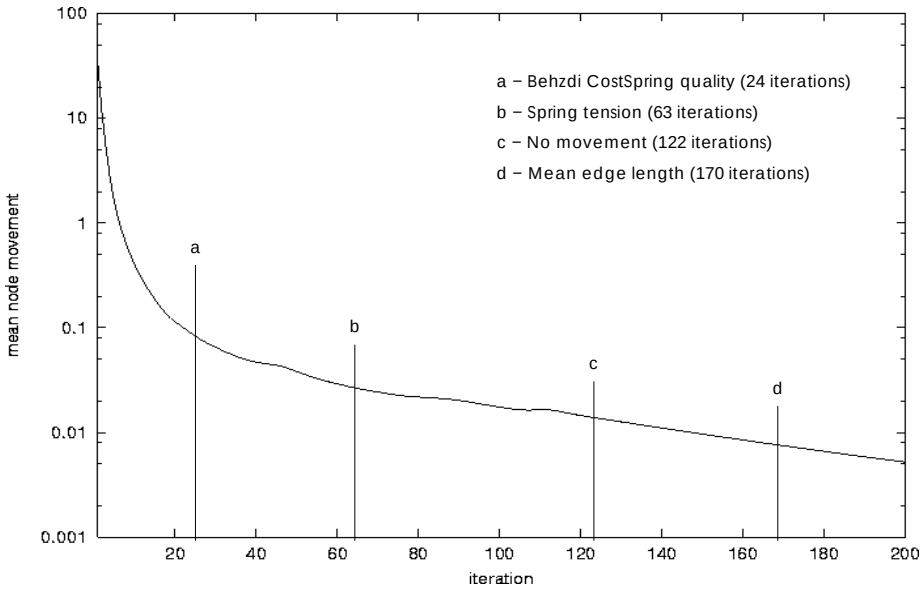
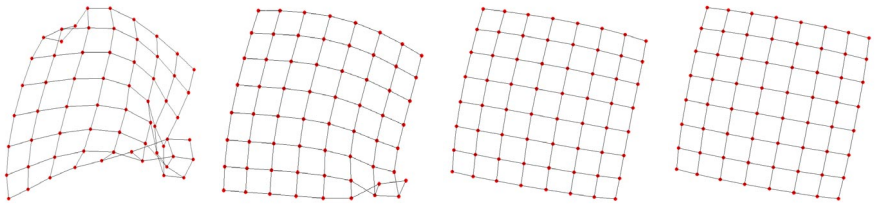


Figure 5.1: Termination points for four terminators on a 64 node square grid



(a) Behzadi quality (b) Spring tension (c) No movement (d) Edge length

Figure 5.2: Layouts obtained from CC at the termination points of Figure 5.1

.

6 The Big Bang model

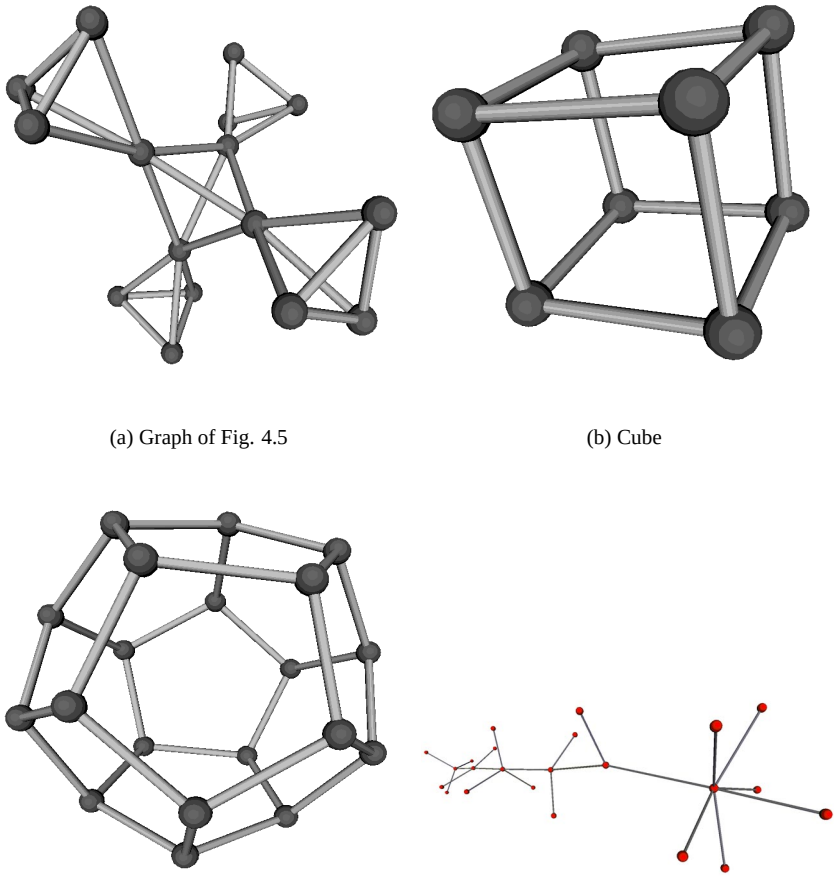
Much of the previous work on FDP systems has been aimed at overcoming layout problems due to unfavourable initial positions of nodes. Tunkelang [36]; Gajer et al. [15] and others have used the idea of adding nodes to the layout one at a time in some specific order, effectively overriding any initial positions. Systems such as Fruchterman and Reingold [14] and Eades [9] largely ignore the problem, resulting in unreliable layout quality for random input data. Davidson and Harel [8] and Kamada and Kawai [25] are not strictly FDP methods and appear less sensitive to initial positions. Their process of analysing the energy minimisation before each node is moved has the apparent effect of reducing dependence on initial positions, but at the cost of significant 'wastage' of calculation and greatly increased computation.

When presented with a favourable initial layout, even the simplest of systems produces excellent results. What is required is a computationally cheap and robust method of presenting an FDP system with a favourable starting point. Our attempt to achieve this goal consists of adding an initial phase to the layout algorithm that differs only in its force model. Specifically we use a different repulsive force formula during the early part of the algorithm's execution. The aim during this first phase is to move nodes away from each other where the direction of movement is determined by universal repulsion and the distance moved at each iteration is a constant value large enough to overcome attraction due to the springs. The result is an expansive early phase that we refer to as the 'Big Bang' in analogy to the ballistic expansion that occurred at the birth of our universe.

During the Big Bang phase, the broad spatial relationships of the nodes are established and any potential 'psychopathic' behaviour is removed. With larger graphs in particular, initial arrangements that produce twists in the shape or the 'traps' discussed previously are eliminated. At the end of the Big Bang phase the graph should be in a favourable starting arrangement from which a simple force model can produce a 'nice' layout. Use of a Big Bang phase should produce several consequences:

- Independence of initial conditions; good layouts should ensue from any starting arrangement,
- reduction in the variance of the number iterations required,
- smoother convergence to a final state, and
- graphs that consistently lay out well *without* a Big Bang phase should lay out at least as well with a Big Bang phase.

All of the preceding consequences have been observed to occur. With correct calibration of the number of Big Bang iterations (I_{bb}) and the size of the Big Bang repulsive force (K_{bb}), even small, regular graphs that routinely give good results are improved



(a) Graph of Fig. 4.5

(b) Cube

(c) Dodecahedron

(d) Tree

Figure 6.1: 3D layouts using the Big Bang

upon. Graphs that have proven to be difficult to lay out well from random starting positions consistently give 'nice' layouts when an appropriate Big Bang phase is applied.

Figure 6.1 shows the graphs of Figure 4.8 as laid out with the application of a Big Bang phase. Of note is that the tree of Figure 4.8(d) has a much straighter spine and the dodecahedron from Figure 4.8(c) is perfectly regular. A slight improvement in the five tetrahedron graph occurs in that the alignment of the outer tetrahedra is consistently more regular. There is no visual improvement in the cube, it was very good anyway, but the layout can be produced in slightly fewer iterations with a well calibrated Big Bang .

Some examples of 'difficult' 3D graphs appear in Figure 6.2, compared with non-Big Bang versions. These are larger graphs and it can clearly be seen that a standard FDP algorithm cannot reliably lay these out well. Using a Big Bang phase we can achieve a near perfect layout every time, regardless of starting arrangement.

Table 6.1 presents a comparison of layout times (number of iterations) with and without a Big Bang phase for a number of graphs. The figures quoted were generated from the mean values over twenty runs of each model and graph combination, using random starting positions. The data show a reduction in the total number of iterations overall ($P[t_{stat} > t_9] < 0.10$), and a significant reduction in the variability of layout times as evidenced by the smaller standard deviations. The effect is more pronounced for larger and more complex graphs. Figure 6.3(a) gives a graphical comparison of the layout times and Figure 6.3(b) a comparison of the standard deviations over the twenty runs sampled for each model.

In Figure 6.4 we present a comparison of activity (defined as mean node movement in each iteration) during layout of the dodecahedron graph between the CC model and the Big Bang. Note the change in the range of the vertical axis between the three charts. Figure 6.4(a) ranges from 0 to 35, Figure 6.4(b) from 0 to 0.6 and Figure 6.4(c) from 0 to 0.016. For the first 40 iterations (the Big Bang phase) no significant difference between the two models occurs. Although from 17 iterations onwards the Big Bang is showing more movement than CC.

At the conclusion of the Big Bang phase, the Big Bang model reverts to the CC force model and we see a dramatic increase in activity as CC begins its layout of the Big Big Bang's output. Finally the Big Bang converges fairly smoothly to finish after 90 iterations. The CC model however, shows far more erratic behaviour and finally

	CC		Big Bang		
Graph (N, E)	mean	std dev	I_{bb}	mean	std dev
Tetrahedron (4, 6)	17	6	4	18	6
Cube (8, 12)	39	6	8	33	8
Dodecahedron (20, 30)	70	10	40	72	9
Tree (fig. 4.6) (17, 16)	252	67	17	253	47
8x8 rectangular grid (64, 112)	343	76	64	221	25
4x16 cylinder (64, 124)	628	216	128	561	12
Hexagonal grid (96, 133)	596	150	96	423	26
20x5 cylinder (100, 180)	292	100	100	331	34
5x20 cylinder (100, 195)	1084	204	300	1027	10
5x20 rectangular grid (100, 175)	1135	282	200	882	16

Table 6.1: Performance comparison of standard CC and Big Bang

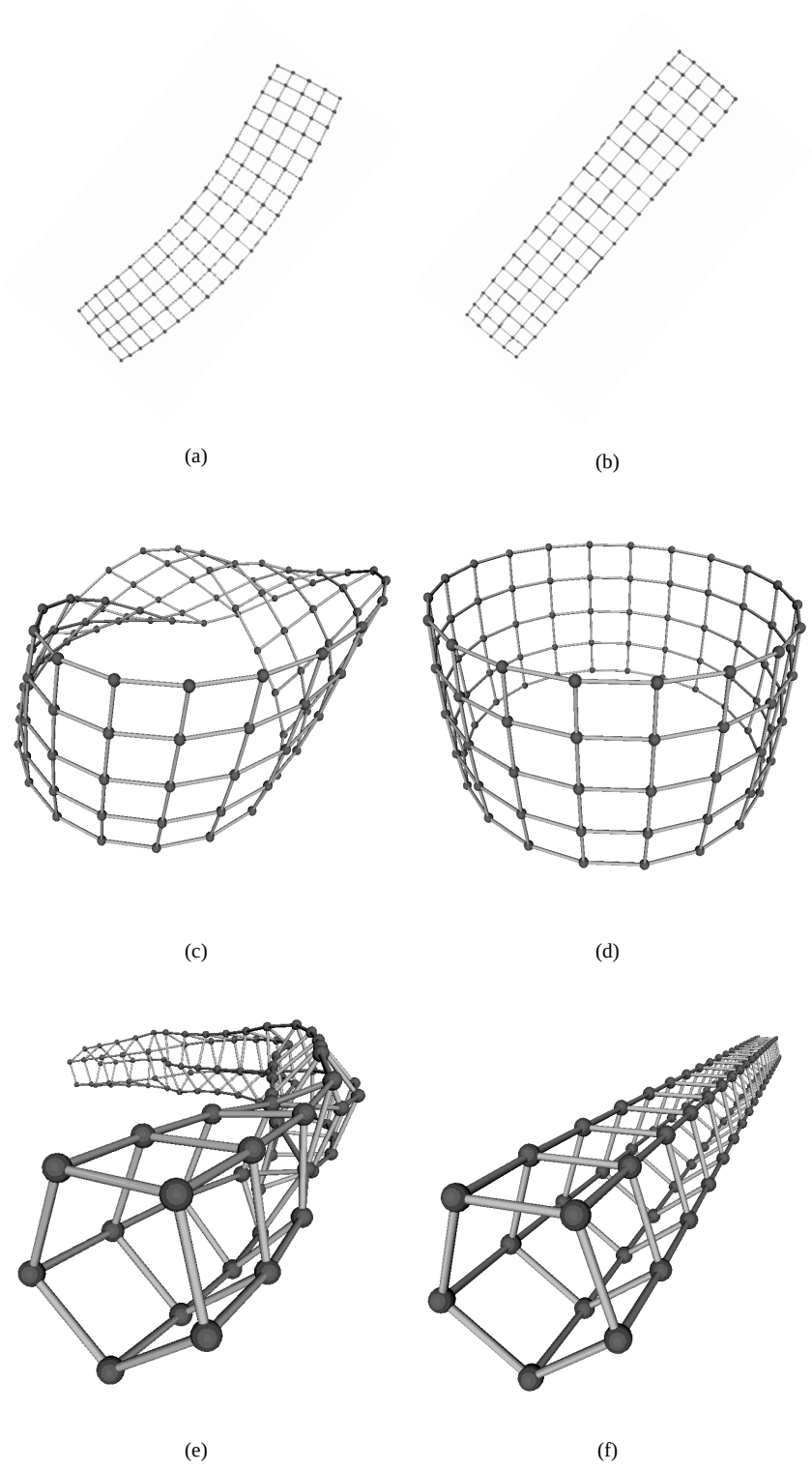
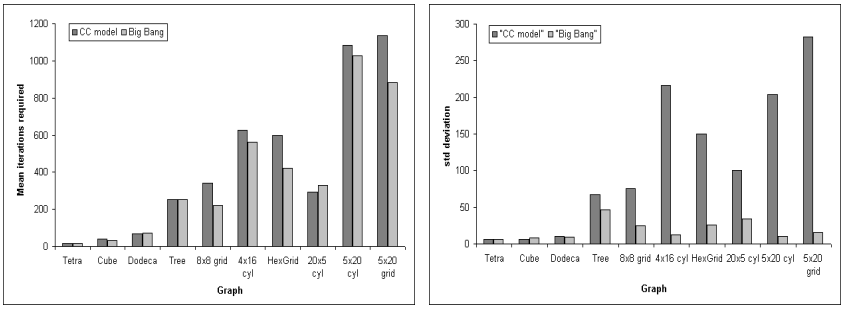


Figure 6.2: 3D layouts of 100 node graphs: without Big Bang: (a, c, e), and with Big Bang: (b, d, f)



(a) (b)

Figure 6.3: Comparisons of time taken and std. deviations for CC and Big Bang

converges to the stopping point after a total of 107 iterations.

6.1 Calibrating the Big Bang

The repulsive force constant (K_{bb}) during the Big Bang phase needs to be high enough to produce expansion in the early stages for maximum effectiveness. The value required for the constant depends on the initial amount of stretch in the springs (roughly equivalent to the density of the nodes in the layout area or volume), and the mean connectivity of the graph. The repulsive force should not be too strong since excessive explosion of the graph will only serve to increase the time taken to shrink it back to size. A suitable setting for the Big Bang force can be obtained using the formula of Equation 6.1. This sets K_{bb} to produce expansion in edges of greater than mean length and appears to give good results, although more experimentation is required. Note that, if generating random arrangements as starting positions, using a smaller bounding box (or cube) will have the same effect as increasing the Big Bang force. Therefore, in a fixed size layout area (volume) smaller graphs will typically require greater values of K_{bb}.

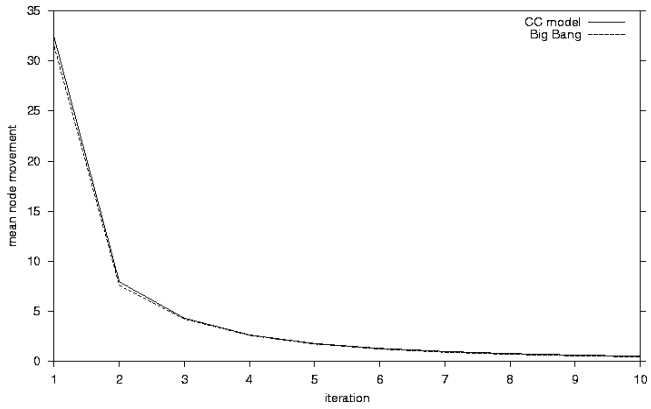
$$K_{bb} = F_a[\frac{1}{|N|}\sum_{e \in E} length(e)] \tag{6.1}$$

Operating the Big Bang repulsive force for an appropriate number of iterations is also important. Not enough Big Bang, and the graph may still be in an unfavourable state, too much, and we waste iterations. The optimal duration of the Big Bang phase is expected to be proportional to the number of nodes in the graph, the mean connectivity, and the 'hardness' of the graph. By hardness we refer to the degree by which a graph, in its laid out form, differs from a compact form, i.e. a flat square for 2D layouts, a dense cube for 3D. A 'hard' graph is either elongated severely, or lays out with 'cavities' in the final shape. A 3D layout with all nodes on the surface of a regular solid is a relatively 'easy' graph, a short cylinder is a 'hard' graph, and a long, narrow, cylinder constitutes a 'very hard' graph.

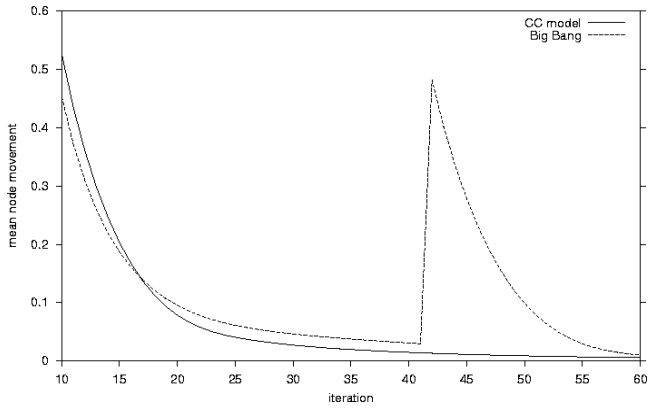
We have not, as yet, reached any conclusions as to an appropriate method of determining the number of Big Bang iterations. As a heuristic we use the number of *edges* in a graph, as this expresses a notion of both the number of nodes and the connectivity. We apply the formula of Equation 6.2 with C equal to 0.5 for easy graphs, 1 for hard graphs and 2 for very hard graphs.

$$I_{bb} = 2C \times |E| \tag{6.2}$$

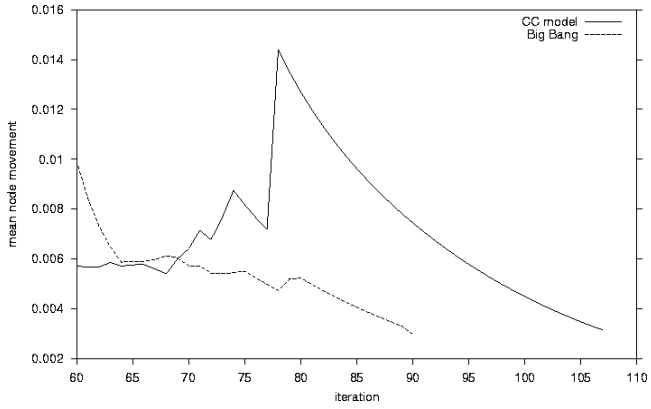
A simpler method for estimating I_{bb} is to multiply the number of nodes by a constant. This rule of thumb is more applicable in that one generally knows the number of nodes in a graph, but not always the number of edges. We have successfully used this rule with a constant value of 6, producing reliably good layouts but incurring penalties in terms of iterations for graphs with low connectivity.



(a) Iterations 1-10, ymax = 35



(b) Iterations 10-60, ymax = 0.6



(c) Iterations 60-110, ymax = 0.016

Figure 6.4: Mean movement by iteration trace for a Dodecahedron layout

7 Conclusions

7.1 ANGLE

ANGLE has proven a useful tool, meeting all of its original requirements. It, and the Big Bang, have been applied successfully as a practical 3D layout tool, forming part of the 'Visualisation Pipeline' described in [23].

The use of Java and its ability to 'bolt on' extra classes without re-compiling existing code is of major benefit for experimental software of this type. Similarly the ability to run the same compilation on various platforms has also proven of benefit. ANGLE was developed under the Solaris operating system and has been successfully run on both Linux and Windows 2000 without needing to change or recompile any class files. Version control has also been made easier by only needing one working copy of the source code.

Additionally the use of the Javadoc utility for documentation 'on the fly' has allowed us to produce software that we consider others could readily learn with minimal effort. Javadoc also ensures that both code and documentation are kept in the one place (in the same file, in fact) thus further aiding distribution to others and reducing the chances of misplacing of mismatching files.

7.2 Force directed placement

Using the software to experiment with algorithms and force models has made it clear that good results can be achieved by quite simple FDP implementations provided they are properly calibrated. In light of our results it appears that much of the additional complexity of algorithms evolved since 1983 may not be necessary if we seek only reasonable layouts of small to medium sized graphs.

7.3 The Big Bang

The addition of a Big Bang phase has given us a robust and reliable version of an FDP implementation. It has addressed all of the points we raised in Section 4 and shows considerable promise as an embedder for generating speedy and effective layouts in two or three dimensions. As well as overcoming the dependence of FDP on initial arrangements, it provides a smooth progression towards a reliably good quality layout in reasonable time. The smooth behaviour also aids the termination problem and allows users to make early terminations where quality can be reduced in favour of speed.

The Big Bang does *no worse* than other methods and is usually significantly better, either in terms of speed (number of iterations), of (aesthetic) quality of layout, or of both.

8 Future work

8.1 ANGLE

We need an improved parser for NGML and enhancements to the language itself. As mentioned in Section 3.6 the upgrading of NGML to a full XML language is desirable. Some of our further work on the Big Bang will require a more powerful graph definition language than the present NGML.

ANGLE itself could be improved in several ways, as follows:

- Implement a force model interface and implement individual force models in their own classes. At present ANGLE requires the force model to be implemented as part of the algorithm. This may prove impractical in that the increased number of method calls in computing a layout may make execution too slow, but such an implementation would allow experimenters to 'mix and match' various force models and algorithms quickly and easily.

- Build more input parsers. At present our experimentation has been limited to the graphs we could readily hand code in NGML, or that could be converted to NGML with the limited range of tools available. ANGLE outputs VRML at present. It would be useful if it could read it also. Additionally, the application could easily be made to support some of the graph markup languages that have been devised by others. This would give us more access to 'standard' graphs used by others to evaluate layout methods.

- Implementing a command line version that can be used as a UNIX pipe. Such an implementation could then be used to parse graph data from other applications and write laid out graphs to file or to a display package. This would provide the ability to test layout methods in 'real' environments.

- Building an interface to the AKAROA multiple replication simulator [12]. This would allow us to gather set precision statistical data on embedder performance in an efficient manner. Currently a Java interface for AKAROA is under development and we should be able to create classes for ANGLE that will attach directly to it.

8.2 Big Bang

Calibration

As mentioned in Section 6.1 we do not have a rigorous means of selecting a suitable Big Bang force constant, and only a rough guide for setting an appropriate number of iterations of the Big Bang phase. More experimentation and some theoretical research should solve the first of these. Determining the iteration count, however, will require a

suitable means of defining the 'hardness' of graphs. Provided we can find computationally cheap methods of determining these parameters we should be able to implement a 'self calibrating' Big Bang that produces optimal results for any graph.

Force model enhancements

The present Big Bang, as with most FDP methods, only operates with graphs where all edges (springs) are identical and also all nodes. For a practical graph layout system we need to be able to specify springs of different strength (representing different relationships) and also nodes with various attributes. We have done some experiments using node attributes, treating them as massive bodies that respond to a unidirectional gravitational force. This enhancement shows promise for laying out trees, for example, and producing other hierarchical structures. An alternative approach is to use a magnetic field-like force that interacts with edges and attempts to align them in a particular direction (see Eades et al. [10,Chapt. 10]).

Final phase

Adding a third phase to the Big Bang may improve its performance. The present system tends to produce a long period in the latter part of the layout process where node movements are small at each iteration. During this period the final layout is generally easy to discern by eye and should be predictable by a good model. We have considered increasing the attractive force towards the latter part of the layout process, since, after a Big Bang start, the attractive force is dominant in determining the result. Some experiments we have run show early promise, but much more work is needed in this area.

Computational complexity

Applying some form of locality to repulsive force calculations can reduce the computational complexity (see Section 4.2). This is essential if we wish to apply FDP, and specifically the Big Bang, to graphs with more than a few hundred nodes. Various means have been tried, typically without significantly reducing the quality of layout produced. Implementation of methods such as that proposed by Barnes and Hut [1] may improve the performance of the Big Bang system. Tunkelang [37] approaches graph drawing as a numerical optimisation problem. His work is based on FDP and contains a number of suggestions that may be usefully applied to the Big Bang.

Appendix A
Upcoming publication

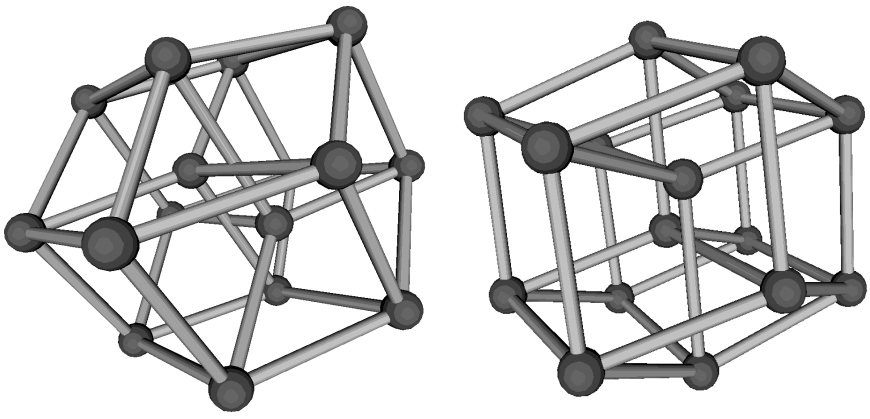
The following 10 pages consist of a pre-print of a paper to appear at invis.au, the Australian Symposium on Information Visualisation, in December, 2001. The paper was written in July-August and details the state of our work at that time.

(Actually the paper is NOT included in this copy of the report, it is accessible at http://www.cosc.canterbury.ac.nz/research/RG/svg/angle/churcher-creek.pdf)

Appendix B
CC and Big Bang layouts

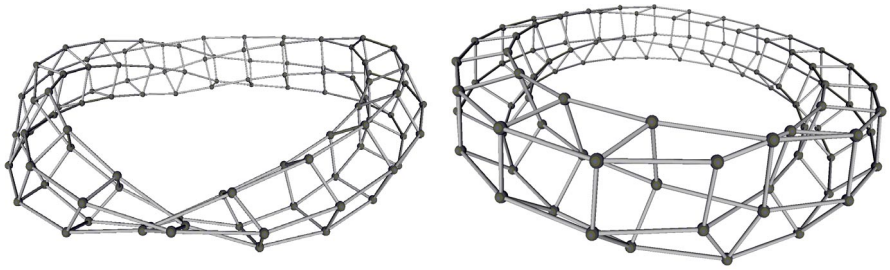
Layouts produced by CC and Big Bang for a selection of graphs. The Hypercube is a common test subject for this purpose. The toroid is a particularly difficult graph to lay out well, and the others are examples from real applications. Figure B.3 is an example of a cyclomatic complexity graph from the original paper on the topic by McCabe. Figure B.4 is a parse tree produced from compiling a Java implementation of 'Hello World'. The graph of Figure B.5 was compiled from a web browser trail of actual browser navigation, and that of Figure B.6 is a representation of the structure of a Java class from Fig. 12 of Irwin and Churcher [23].

In some cases there is little qualitative difference between the CC and Big Bang layouts. However, in the cases of the cyclomatic complexity graph and the web trail the Big Bang model reduces the number of iterations required. Also the quality of layout for the web trail and the hypercube is noticeably better when viewed in true 3D with a VRML browser. The parse tree is marginally better in 3D layout, but probably not enough to warrant the extra iterations used. Typically, we have found that the CC model works very well with trees, and other very sparse graphs, and the Big Bang does little to improve the results.



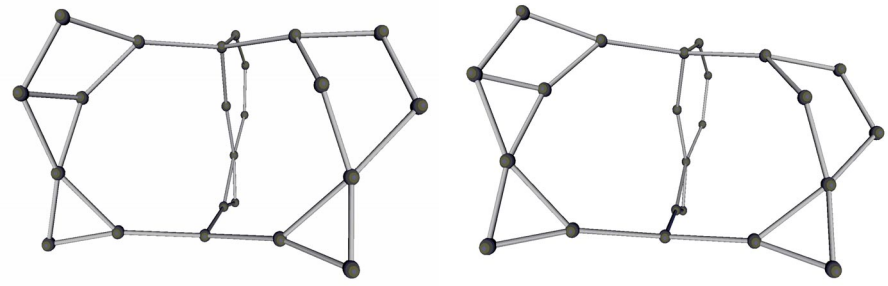
(a) CC–26 iterations (b) Big Bang–39 iterations

Figure B.1: Hypercube



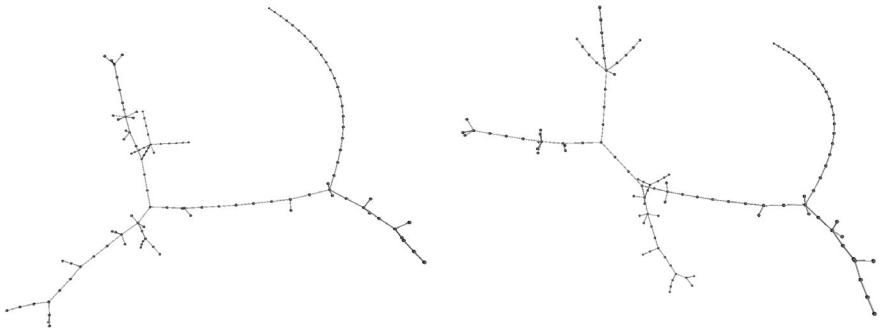
(a) CC–467 iterations (b) Big Bang–737 iterations

Figure B.2: 100 node toroid



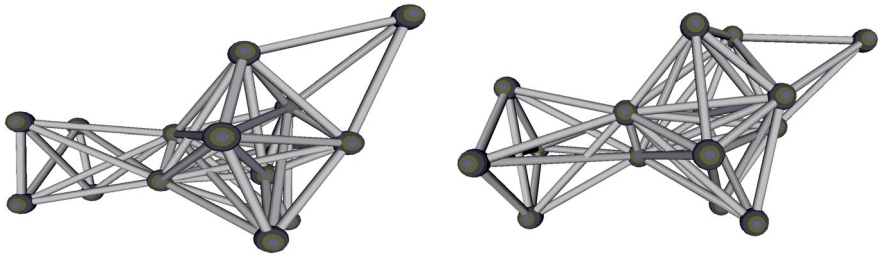
(a) CC–362 iterations (b) Big Bang–327 iterations

Figure B.3: Cyclomatic complexity graph



(a) CC–3419 iterations (b) Big Bang–4659 iterations

Figure B.4: Parse tree



(a) CC–159 iterations (b) Big Bang–111 iterations

Figure B.5: World-wide-web navigation graph

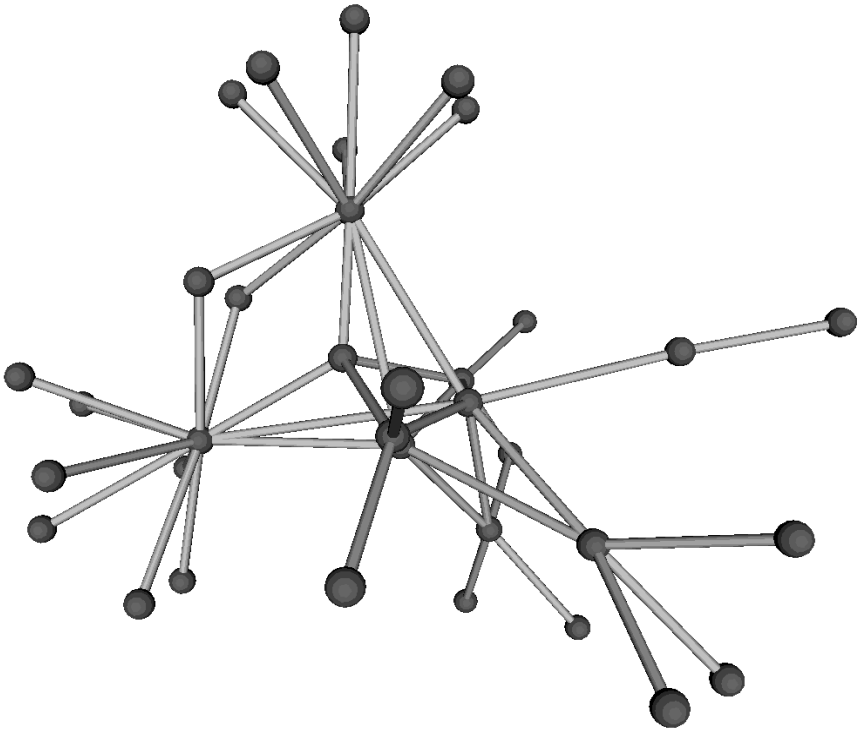


Figure B.6: Graph of the structure of a Java class, laid out with the Big Bang

References

[1] Josh Barnes and Piet Hut. A hierarchical $\mathcal{O}(NlogN)$ force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[2] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, 1994.

[3] Lila Behzadi. An improved spring-based graph embedding algorithm and LayoutShow: a Java environment for graph drawing. Master's thesis, York University, North York, Ontario, Canada, 1999.

[4] Lila Behzadi. Layoutshow: A signed applet/application for graph drawing and experimentation. In Jan Kratochvíl, editor, *Graph Drawing: 7th International Symposium, GD '99, Proceedings*, number 1731 in Lecture Notes in Computer Science, pages 242–249, Štiřín Castle, Czech Republic, September 1999. Springer-Verlag.

[5] Rikk Carey and Gavin Bell. The annotated VRML 97 reference. URL: `http://www.best.com/~rikk/Book`, 1999.

[6] Neville Churcher and Alan Creek. Building virtual worlds with the Big Bang model. In *invis.au: Australian Symposium on Information Visualisation*, Sydney, Australia, December 2001. Accepted for publication.

[7] Robert F. Cohen, Peter Eades, Tao Lin, and Frank Ruskey. Three-dimensional graph drawing. In Tamassia and Tollis [35], pages 1–11.

[8] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics (TOG)*, 15(4):301–331, October 1996.

[9] Peter Eades. A heuristic for graph drawing. In D. S. Meek and G. H. J. van Rees, editors, *Proceedings of the 13th Manitoba Conference on Numerical Mathematics and Computing*, Winnipeg, Canada, January 1983. Utilitas Mathematica Publishing.

[10] Peter Eades, Guiseppe Di Battista, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualisation of Graphs*. Prentice Hall, New Jersey, 1999.

[11] Peter Eades, Mao Ling Huang, and Junhu Wang. Online animated graph drawing using a modified spring algorithm. Technical Report 5, University of Newcastle, Australia, 1997.

[12] Greg Ewing, Krys Pawlikowski, and Don McNickle. AKAROA2 user manual. URL: `http://www.cosc.canterbury.ac.nz/research/RG/net_sim/simulation_g% roup/akaroa/download.html`, 2001.

[13] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs. In Tamassia and Tollis [35], pages 388–403.

[14] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11), 1991.

[15] P. Gajer, M. Goodrich, and S. Kobourov. A fast multi-dimensional algorithm for drawing large graphs. In Joe Marks, editor, *Graph Drawing: 8th International Symposium, GD 2000, Proceedings*, number 1984 in Lecture Notes in Computer Science, pages 211–221, Colonial Williamsburg, Va., USA, September 2000. Springer-Verlag.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Ma., USA, 1995.

[17] Emden R. Gansner and Stephen C. North. Improved force-directed layouts. In Sue H. Whitesides, editor, *Graph Drawing: 6th International Symposium, GD '98, Proceedings*, number 1547 in Lecture Notes in Computer Science, pages 364–373, Montréal, Canada, August 1998. Springer-Verlag.

[18] Joseph Gil and Stuart Kent. Three dimensional software modelling. In *Proceedings of the International Conference on Software Engineering*, pages 105–114, Kyoto, Japan, April 1998. Association for Computing Machinery.

[19] Martin Gogolla, Oliver Radfelder, and Mark Richters. Towards three-dimensional representation and animation of UML diagrams. In *Proceedings of UML'99*, Fort Collins, Colorado, USA, October 1999. IEEE Computer Society Press.

[20] I. Herman and M.S. Marshall. GraphXML – an XML based graph interchange format. Technical Report INS-R0009, ISSN 1386-3681, National Research Institute for Mathematics and Computer Science (CWI), Amsterdam, The Netherlands, 2000.

[21] Michael Himsolt. GraphEd: a graphical platform for the implementation of graph algorithms. In Tamassia and Tollis [35], pages 182–193.

[22] Scott E. Hudson. CUP user's manual, 1999. URL: `http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html`

[23] Warwick Irwin and Neville Churcher. XML in the visualisation pipeline. Technical Report 05/01, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, 2001.

[24] Dean F. Jerding and John T. Stasko. Using visualisation to foster object-oriented program understanding. Technical Report GIT-GVU-94-33, Georgia Institute of Technology, Atlanta, Ga., 1994.

[25] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[26] Gerwin Klein. Jflex user's manual, 2001.
URL: `http://www.jflex.de/manual.html`

[27] Hideki Koike and Hui-Chu Chu. How does 3-D visualization work in software engineering? : Emprirical study of a 3-D version/module visualization system. In *International Conference on Software Engineering (ICSE), 1998*, Kyoto, Japan, April 1998. IEEE Computer Society Press.

[28] Aruna Kumar and Richard H. Fowler. A spring modeling algorithm to position nodes of an undirected graph in three dimensions. Technical report, University of Texas - Pan American, Edinburg, Texas, USA, 1994.
URL: `http://www.cs.panam.edu/info_vis/spr_tr.html`

[29] John Lewis and William Loftus. *Java Software Solutions: Foundations of Program Design*. Addison Wesley, 1st edition, 1998.

[30] Maurizio Patrignani and Francesco Vargiu. 3DCube: A tool for three dimensional graph drawing. In Giuseppe Di Battista, editor, *Graph Drawing: 5th International Symposium, GD '97, Proceedings*, number 1353 in Lecture Notes in Computer Science, pages 284–290, Rome, Italy, September 1997. Springer-Verlag.

[31] N. Quinn and M. Breur. A force directed component placement procedure for printed circuit boards. *IEEE Trans. Circuits and Systems*, CAS-26(6):377–388, 1979.

[32] Stephen P. Reiss. 3-D visualisation of program information. In Tamassia and Tollis [35], pages 13–24.

[33] Aaron Scott. A survey of graph drawing systems. Technical Report 95-1, University of Newcastle, Sydney, Australia, 1994.

[34] Ian Sommerville. *Software Engineering*. Addison Wesley, Harlow, UK, fifth edition, 1996.

[35] Roberto Tamassia and Ioannis G. Tollis, editors. *Graph Drawing: DIMACS International Workshop, GD'94, Proceedings*, number 894 in Lecture Notes in Computer Science, Princeton, NJ, USA, October 1994. Springer-Verlag.

[36] Daniel Tunkelang. A practical approach to drawing undirected graphs. Technical Report CMU-CS-94-161, Carnegie Mellon University, Pittsburgh, PA, USA, June 1994.

[37] Daniel Tunkelang. *A Numerical Optimization Approach to General Graph Drawing*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, January 1999.

[38] W. T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, 13(3):743–768, 1963.

[39] Colin Ware and Glenn Franck. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Transactions on Graphics*, 15(2): 121–140, 1996.